
Java
8

目錄

Java8新特性探究	0
一、通往lambda之路：语法篇	1
二、深入解析默认方法	2
三、解开lambda最强作用的神秘面纱	3
四、类型注解：复杂还是便捷	4
五、重复注解（repeating annotations）	5
六、泛型的目标类型推断	6
七、深入解析日期和时间：JSR310	7
八、精简的JRE详解	8
九、跟OOM：Permgen说再见吧	9
十、StampedLock将是解决同步问题的新宠	10
十一：Base64详解	11
十二、Nashorn：新犀牛	12
十三：JavaFX8新特性以及开发2048游戏	13

Java8新特性探究

一、通往lambda之路：语法篇

> 来源：Java 8新特性探究（一）通往lambda之路 语法篇

现在开始要灌输一些概念性的东西了，这能帮助你理解lambda更加透彻一点，如果你之前听说过，也可当是温习，所谓温故而知新……

在开始之前，可以同步下载jdk 8 和 IDE，IDE根据个人习惯了，不过Eclipse官方版本还没出来，所以目前看的话，netbean7.4是首选的，毕竟前段子刚刚出的正式版本，以下是他们的下载地址。

- jdk 8：https://jdk8.java.net/download.html （毕竟是国外的网站，如果下载慢，可以到我的云盘下载http://pan.baidu.com/share/link?shareid=61064476&uk=4060588963）
- Netbeans 7.4正式版：https://netbeans.org/downloads/(推荐，oracle官方发布)
- IDEA 12 EAP：http://confluence.jetbrains.net/display/IDEADEV/IDEA+12+EAP
- Unofficial builds of Eclipse：http://downloads.efxclipse.org/eclipse-java8/

函数式接口

函数式接口（functional interface 也叫功能性接口，其实是同一个东西）。简单来说，函数式接口是只包含一个方法的接口。比如Java标准库中的java.lang.Runnable和java.util.Comparator都是典型的函数式接口。java 8提供 @FunctionalInterface作为注解,这个注解是非必须的，只要接口符合函数式接口的标准（即只包含一个方法的接口），虚拟机会自动判断，但 最好在接口上使用注解@FunctionalInterface进行声明，以免团队的其他人员错误地往接口中添加新的方法。

Java中的lambda无法单独出现，它需要一个函数式接口来盛放，lambda表达式方法体其实就是函数接口的实现，下面讲到语法会讲到

Lambda语法

包含三个部分

1. 一个括号内用逗号分隔的形式参数，参数是函数式接口里面方法的参数
2. 一个箭头符号：->
3. 方法体，可以是表达式和代码块，方法体函数式接口里面方法的实现，如果是代码块，则必须用{}来包裹起来，且需要一个return 返回值，但有个例外，若函数式接口里面方法返回值是void，则无需{}

总体看起来像这样

```
(parameters) -> expression 或者 (parameters) -> { statements; }
```

看一个完整的例子，方便理解

```
/**
 * 测试lambda表达式
 *
 * @author benhail
 */
public class TestLambda {

    public static void runThreadUseLambda() {
        //Runnable是一个函数接口，只包含了有个无参数的，返回void的run方法；
        //所以lambda表达式左边没有参数，右边也没有return，只是单纯的打印一句话
        new Thread(() ->System.out.println("lambda实现的线程")).start();
    }

    public static void runThreadUseInnerClass() {
        //这种方式就不多讲了，以前旧版本比较常见的做法
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("内部类实现的线程");
            }
        }).start();
    }

    public static void main(String[] args) {
        TestLambda.runThreadUseLambda();
        TestLambda.runThreadUseInnerClass();
    }
}
```

可以看出，使用lambda表达式设计的代码会更加简洁，而且还可读。

方法引用

其实是lambda表达式的一个简化写法，所引用的方法其实是lambda表达式的方法体实现，语法也很简单，左边是容器（可以是类名，实例名），中间是"::"，右边是相应的方法名。如下所示：

```
ObjectReference::methodName
```

一般方法的引用格式是

1. 如果是静态方法，则是ClassName::methodName。如 Object ::equals
2. 如果是实例方法，则是Instance::methodName。如Object obj=new Object();obj::equals;
3. 构造函数.则是ClassName::new

再来看一个完整的例子，方便理解

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import javax.swing.JButton;
import javax.swing.JFrame;

/**
 *
 * @author benhail
 */
public class TestMethodReference {

    public static void main(String[] args) {

        JFrame frame = new JFrame();
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);

        JButton button1 = new JButton("点我!");
        JButton button2 = new JButton("也点我!");

        frame.getContentPane().add(button1);
        frame.getContentPane().add(button2);
        //这里addActionListener方法的参数是ActionListener，是一个函数式接口
        //使用lambda表达式方式
        button1.addActionListener(e -> { System.out.println("这里是Lambda实现方式"); });
        //使用方法引用方式
        button2.addActionListener(TestMethodReference::doSomething);

    }
    /**
     * 这里是函数式接口ActionListener的实现方法
     * @param e
     */
    public static void doSomething(ActionEvent e) {

        System.out.println("这里是方法引用实现方式");

    }
}
```

可以看出，doSomething方法就是lambda表达式的实现，这样的好处就是，如果你觉得lambda的方法体会很长，影响代码可读性，方法引用就是个解决办法

总结

以上就是lambda表达式语法的全部内容了，相信大家对lambda表达式都有一定的理解了，但只是代码简洁了这个好处的话，并不能打动很多观众，java 8也不会这么令人期待，其实java 8引入lambda迫切需求是因为lambda 表达式能简化集合上数据的多线程或者多核的处理，提供更快的集合处理速度 ，这个后续会讲到，关于JEP126的这一特性，将分3部分，之所以分开，是因为这一特性可写的东西太多了，这部分让读者熟悉lambda表达式以及方法引用的语法和概念，第二部分则是虚拟扩展方法（default method）的内容，最后一部分则是大数据集合的处理，解开lambda表达式的最强作用的神秘面纱。敬请期待。。。。

二、深入解析默认方法

> 来源：Java 8新特性探究（二）深入解析默认方法

上篇讲了 lambda表达式的语法，但只是 JEP126 特性的一部分，另一部分就是默认方法（也称为虚拟扩展方法或防护方法）

什么是默认方法，为什么要有默认方法

简单说，就是接口可以有实现方法，而且不需要实现类去实现其方法。只需在方法名前面加个default关键字即可。

为什么要有这个特性？首先，之前的接口是个双刃剑，好处是面向抽象而不是面向具体编程，缺陷是，当需要修改接口时候，需要修改全部实现该接口的类，目前的java 8之前的集合框架没有foreach方法，通常能想到的解决办法是在JDK里给相关的接口添加新的方法及实现。然而，对于已经发布的版本，是没法在给接口添加新方法的同时不影响已有的实现。所以引进的默认方法。他们的目的是为了解决接口的修改与现有的实现不兼容的问题。

简单的例子

一个接口A，Clazz类实现了接口A。

```
public interface A {
    default void foo(){
       System.out.println("Calling A.foo()");
    }
}

public class Clazz implements A {
    public static void main(String[] args){
       Clazz clazz = new Clazz();
       clazz.foo();//调用A.foo()
    }
}
```

代码是可以编译的，即使Clazz类并没有实现foo()方法。在接口A中提供了foo()方法的默认实现。

java 8抽象类与接口对比

这一个功能特性出来后，很多同学都反应了，java 8的接口都有实现方法了，跟抽象类还有什么区别？其实还是有的，请看下表对比。。

相同点	不同点
1.都是抽象类型；	1.抽象类不可以多重继承，接口可以（无论是多重类型继承还是多重行为继承）；
2.都可以有实现方法（以前接口不行）；	2.抽象类和接口所反映出的设计理念不同。其实抽象类表示的是”is-a”关系，接口表示的是”like-a”关系；
3.都可以不需要实现类或者继承者去实现所有方法，（以前不行，现在接口中默认方法不需要实现者实现）	3.接口中定义的变量默认是public static final 型，且必须给其初值，所以实现类中不能改变其值；抽象类中的变量默认是 friendly 型，其值可以在子类中重新定义，也可以重新赋值。

多重继承的冲突说明

由于同一个方法可以从不同接口引入，自然而然的会有冲突的现象，默认方法判断冲突的规则如下：

1.一个声明在类里面的方法优先于任何默认方法（classes always win）

2.否则，则会优先选取最具体的实现，比如下面的例子 B重写了A的hello方法。

```
public interface A {
    default void hello() { System.out.println("Hello World from A"); }
}
public interface B extends A {
    default void hello() { System.out.println("Hello World from B"); }
}
public class C implements B, A {
    public static void main(String... args) {
        new C().hello();
    }
}
```



输出结果是：Hello World from B

如果想调用A的默认函数，则用到新语法X.super.m(…),下面修改C类，实现A接口，重写一个hello方法，如下所示：

```
public class C implements A{

    @Override
    public void hello(){
        A.super.hello();
    }

    public static void main(String[] args){
        new C().hello();
    }
}
```

输出结果是：Hello World from A

总结

默认方法给予我们修改接口而不破坏原来的实现类的结构提供了便利，目前java 8的集合框架已经大量使用了默认方法来改进了，当我们最终开始使用Java 8的lambdas表达式时，提供给我们一个平滑的过渡体验。也许将来我们会在API设计中看到更多的默认方法的应用。

跟上篇博文结合起来，就是JEP126的全部了，后面还有54个特性等着我们去探究，为了让大家比较深刻了解lambda，学以致用，下一篇还是lambda的内容，预告一下下篇的标题：《Java 8特性探究（三）解开lambda表达式最强作用的神秘面纱》，第二个特性 将从第四篇开始，谢谢大家支持，敬请期待。。。

三、解开lambda最强作用的神秘面纱

> 来源：Java 8新特性探究（三）解开lambda最强作用的神秘面纱

我们期待了很久lambda为java带来闭包的概念，但是如果我们不在集合中使用它的话，就损失了很大价值。现有接口迁移成为lambda风格的问题已经通过default methods解决了，在这篇文章将深入解析Java集合里面的批量数据操作（bulk operation），解开lambda最强作用的神秘面纱。

1.关于JSR335

JSR是Java Specification Requests的缩写，意思是Java 规范请求,Java 8 版本的主要改进是Lambda 项目（JSR 335），其目的是使 Java 更易于为多核处理器编写代码。JSR 335=lambda表达式+接口改进（默认方法）+批量数据操作。加上前面两篇，我们已是完整的学习了JSR335的相关内容了。

2.外部VS内部迭代

以前Java集合是不能够表达内部迭代的，而只提供了一种外部迭代的方式，也就是for或者while循环。

```
List persons = asList(new Person("Joe"), new Person("Jim"), new Person("John"));
for (Person p :  persons) {
   p.setLastName("Doe");
}
```

上面的例子是我们以前的做法，也就是所谓的外部迭代，循环是固定的顺序循环。在现在多核的时代，如果我们想并行循环，不得不修改以上代码。效率能有多大提升还说定，且会带来一定的风险（线程安全问题等等）。

要描述内部迭代，我们需要用到Lambda这样的类库,下面利用lambda和Collection.forEach重写上面的循环

```
persons.forEach(p->p.setLastName("Doe"));
```

现在是由jdk 库来控制循环了，我们不需要关心last name是怎么被设置到每一个person对象里面去的，库可以根据运行环境来决定怎么做，并行，乱序或者懒加载方式。这就是内部迭代，客户端将行为p.setLastName当做数据传入api里面。

内部迭代其实和集合的批量操作并没有密切的联系，借助它我们感受到语法表达上的变化。真正有意思的和批量操作相关的是新的流（stream）API。新的java.util.stream包已经添加进JDK 8了。

3.Stream API

流（Stream）仅仅代表着数据流，并没有数据结构，所以他遍历完一次之后便再也无法遍历（这点在编程时候需要注意，不像Collection，遍历多少次里面都还有数据），它的来源可以是Collection、array、io等等。

3.1中间与终点方法

流作用是提供了一种操作大数据接口，让数据操作更容易和更快。它具有过滤、映射以及减少遍历数等方法，这些方法分两种：中间方法和终端方法，“流”抽象天生就该是持续的，中间方法永远返回的是Stream，因此如果我们要获取最终结果的话，必须使用终点操作才能收集流产生的最终结果。区分这两个方法是看他的返回值，如果是Stream则是中间方法，否则是终点方法。具体请参照Stream的api。

简单介绍下几个中间方法（filter、map）以及终点方法（collect、sum）

3.1.1Filter

在数据流中实现过滤功能是首先我们可以想到的最自然的操作了。Stream接口暴露了一个filter方法，它可以接受表示操作的Predicate实现来使用定义了过滤条件的lambda表达式。

```
List persons = …
Stream personsOver18 = persons.stream().filter(p -> p.getAge() > 18);//过滤18岁以上的人
```

3.1.2Map

假使我们现在过滤了一些数据，比如转换对象的时候。Map操作允许我们执行一个Function的实现（Function的泛型T,R分别表示执行输入和执行结果），它接受入参并返回。首先，让我们来看看怎样以匿名内部类的方式来描述它：

```
Stream adult= persons
              .stream()
              .filter(p -> p.getAge() > 18)
              .map(new Function() {
                  @Override
                  public Adult apply(Person person) {
                     return new Adult(person);//将大于18岁的人转为成年人
                  }
              });
```

现在，把上述例子转换成使用lambda表达式的写法：

```
Stream map = persons.stream()
                    .filter(p -> p.getAge() > 18)
                    .map(person -> new Adult(person));
```

3.1.3Count

count方法是一个流的终点方法，可使流的结果最终统计，返回int，比如我们计算一下满足18岁的总人数

```
int countOfAdult=persons.stream()
                       .filter(p -> p.getAge() > 18)
                       .map(person -> new Adult(person))
                       .count();
```

3.1.4Collect

collect方法也是一个流的终点方法，可收集最终的结果

```
List adultList= persons.stream()
                       .filter(p -> p.getAge() > 18)
                       .map(person -> new Adult(person))
                       .collect(Collectors.toList());
```

或者，如果我们想使用特定的实现类来收集结果：

```
List adultList = persons
                 .stream()
                 .filter(p -> p.getAge() > 18)
                 .map(person -> new Adult(person))
                 .collect(Collectors.toCollection(ArrayList::new));
```

篇幅有限，其他的中间方法和终点方法就不一一介绍了，看了上面几个例子，大家明白这两种方法的区别即可，后面可根据需求来决定使用。

3.2顺序流与并行流

每个Stream都有两种模式：顺序执行和并行执行。

顺序流：

```
List <Person> people = list.getStream.collect(Collectors.toList());
```

并行流：

```
List <Person> people = list.getStream.parallel().collect(Collectors.toList());
```

顾名思义，当使用顺序方式去遍历时，每个item读完后再读下一个item。而使用并行去遍历时，数组会被分成多个段，其中每一个都在不同的线程中处理，然后将结果一起输出。

3.2.1并行流原理：

```
List originalList = someData;
split1 = originalList(0, mid);//将数据分小部分
split2 = originalList(mid,end);
new Runnable(split1.process());//小部分执行操作
new Runnable(split2.process());
List revisedList = split1 + split2;//将结果合并
```

大家对hadoop有稍微了解就知道，里面的 MapReduce 本身就是用于并行处理大数据集的软件框架，其 处理大数据的核心思想就是大而化小，分配到不同机器去运行map，最终通过reduce将所有机器的结果结合起来得到一个最终结果，与MapReduce不同，Stream则是利用多核技术可将大数据通过多核并行处理，而MapReduce则可以分布式的。

3.2.2顺序与并行性能测试对比

如果是多核机器，理论上并行流则会比顺序流快上一倍，下面是测试代码

```
long t0 = System.nanoTime();

//初始化一个范围100万整数流,求能被2整除的数字，toArray（）是终点方法

int a[]=IntStream.range(0, 1_000_000).filter(p -> p % 2==0).toArray();

long t1 = System.nanoTime();

//和上面功能一样，这里是用并行流来计算

int b[]=IntStream.range(0, 1_000_000).parallel().filter(p -> p % 2==0).toArray();

long t2 = System.nanoTime();

//我本机的结果是serial: 0.06s, parallel 0.02s，证明并行流确实比顺序流快

System.out.printf("serial: %.2fs, parallel %.2fs%n", (t1 - t0) * 1e-9, (t2 - t1) * 1e-9);
```

3.3关于**Folk/Join**框架

应用硬件的并行性在java 7就有了，那就是 java.util.concurrent 包的新增功能之一是一个 fork-join 风格的并行分解框架，同样也很强大高效，有兴趣的同学去研究，这里不详谈了，相比Stream.parallel()这种方式，我更倾向于后者。

4.总结

如果没有lambda，Stream用起来相当别扭，他会产生大量的匿名内部类，比如上面的3.1.2map例子，如果没有default method，集合框架更改势必会引起大量的改动，所以lambda+default method使得jdk库更加强大，以及灵活，Stream以及集合框架的改进便是最好的证明。

java 8特性探究系列写了3篇了，作为大餐，将java 8的重量级特性lambda与default method写在前面，下篇上个小菜，荤素搭配，也是语言相关的，JEP104 Java 类型的注解的探究，同时谢谢大家的支持，欢迎提出建议。如果你想了解哪些特性，欢迎给我发留言。

四、类型注解：复杂还是便捷

> 来源：Java 8新特性探究（四）类型注解 复杂还是便捷

本文将介绍java 8的第二个特性：类型注解。

注解大家都知道，从java5开始加入这一特性，发展到现在已然是遍地开花，在很多框架中得到了广泛的使用，用来简化程序中的配置。那充满争议的类型注解究竟是什么？复杂还是便捷？

什么是类型注解

在java 8之前，注解只能是在声明的地方所使用，比如类，方法，属性；java 8里面，注解可以应用在任何地方，比如：

- 创建类实例

```
new @Interned MyObject();
```

- 类型映射

```
myString = (@NonNull String) str;
```

- implements 语句中

```
class UnmodifiableList<T> implements @Readonly List<@Readonly T> { … }
```

- throw exception声明

```
void monitorTemperature() throws @Critical TemperatureException { … }
```

需要注意的是，类型注解只是语法而不是语义，并不会影响java的编译时间，加载时间，以及运行时间，也就是说，编译成class文件的时候并不包含类型注解。

类型注解的作用

先看看下面代码

```
Collections.emptyList().add("One");
int i=Integer.parseInt("hello");
System.console().readLine();
```

上面的代码编译是通过的，但运行是会分别报UnsupportedOperationException；NumberFormatException；NullPointerException异常，这些都是runtime error；

类型注解被用来支持在Java的程序中做强类型检查。配合插件式的check framework，可以在编译的时候检测出runtime error，以提高代码质量。这就是类型注解的作用了。

check framework

check framework是第三方工具，配合Java的类型注解效果就是1+1>2。它可以嵌入到javac编译器里面，可以配合ant和maven使用，也可以作为

```
[Eclipse](http://res.importnew.com/eclipse "Eclipse ImportNew主页")
```

插件。地址是http://types.cs.washington.edu/checker-framework/。 check framework可以找到类型注解出现的地方并检查，举个简单的例子：

```
import checkers.nullness.quals.*;
public class GetStarted {
    void sample() {
        @NonNull Object ref = new Object();
    }
}
```

使用javac编译上面的类

```
javac -processor checkers.nullness.NullnessChecker GetStarted.java
```

编译是通过，但如果修改成

```
@NonNull Object ref = null;
```

再次编译，则出现

```
GetStarted.java:5: incompatible types.
found   : @Nullable <nulltype>
required: @NonNull Object
        @NonNull Object ref = null;
                              ^
1 error
```

如果你不想使用类型注解检测出来错误，则不需要processor，直接javac GetStarted.java是可以编译通过的，这是在java 8 with Type Annotation Support版本里面可以，但java 5,6,7版本都不行，因为javac编译器不知道@NonNull是什么东西，但check framework 有个向下兼容的解决方案，就是将类型注解nonnull用/**/注释起来，比如上面例子修改为

```
import checkers.nullness.quals.*;
public class GetStarted {
    void sample() {
        /*@NonNull*/ Object ref = null;
    }
}
```

这样javac编译器就会忽略掉注释块，但用check framework里面的javac编译器同样能够检测出nonnull错误。 通过类型注解+check framework我们可以看到，现在runtime error可以在编译时候就能找到。

关于JSR 308

JSR 308想要解决在Java 1.5注解中出现的两个问题：

- 在句法上对注解的限制：只能把注解写在声明的地方
- 类型系统在语义上的限制：类型系统还做不到预防所有的bug

JSR 308 通过如下方法解决上述两个问题：

- 对Java语言的句法进行扩充，允许注解出现在更多的位置上。包括：方法接收器（method receivers，译注：例public int size() @Readonly { … }），泛型参数，数组，类型转换，类型测试，对象创建，类型参数绑定，类继承和throws子句。其实就是类型注解，现在是java 8的一个特性
- 通过引入可插拔的类型系统（pluggable type systems）能够创建功能更强大的注解处理器。类型检查器对带有类型限定注解的源码进行分析，一旦发现不匹配等错误之处就会产生警告信息。其实就是check framework

对JSR308，有人反对，觉得更复杂更静态了，比如

```
@NotEmpty List<@NonNull String> strings = new ArrayList<@NonNull String>()>
```

换成动态语言为

```
var strings = ["one", "two"];
```

有人赞成，说到底，代码才是“最根本”的文档。代码中包含的注解清楚表明了代码编写者的意图。当没有及时更新或者有遗漏的时候，恰恰是注解中包含的意图信息，最容易在其他文档中被丢失。而且将运行时的错误转到编译阶段，不但可以加速开发进程，还可以节省测试时检查bug的时间。

总结

并不是人人都喜欢这个特性，特别是动态语言比较流行的今天，所幸，java 8并不强求大家使用这个特性，反对的人可以不使用这一特性，而对代码质量有些要求比较高的人或公司可以采用JSR 308，毕竟代码才是“最基本”的文档，这句话我是赞同的。虽然代码会增多，但可以使你的代码更具有表达意义。对这个特性有何看法，大家各抒己见。。。。

五、重复注解（repeating annotations）

> 来源：Java 8新特性探究（五）重复注解（repeating annotations）

知识回顾

前面介绍了：

- lambda表达式和默认方法 （JEP 126）
- 批量数据操作（JEP 107）
- 类型注解（JEP 104）

注：JEP=JDK Enhancement-Proposal (JDK 增强建议)，每个JEP即一个新特性。

在java 8里面，注解一共有2个改进，一个是类型注解，在上篇已经介绍了，本篇将介绍另外一个注解的改进：重复注解（JEP 120）。

什么是重复注解

允许在同一申明类型（类，属性，或方法）的多次使用同一个注解

一个简单的例子

java 8之前也有重复使用注解的解决方案，但可读性不是很好，比如下面的代码：

```
public @interface Authority {
     String role();
}

public @interface Authorities {
    Authority[] value();
}

public class RepeatAnnotationUseOldVersion {

    @Authorities({@Authority(role="Admin"),@Authority(role="Manager")})
    public void doSomeThing(){
    }
}
```

由另一个注解来存储重复注解，在使用时候，用存储注解Authorities来扩展重复注解，我们再来看看java 8里面的做法：

```
@Repeatable(Authorities.class)
public @interface Authority {
     String role();
}

public @interface Authorities {
    Authority[] value();
}

public class RepeatAnnotationUseNewVersion {
    @Authority(role="Admin")
    @Authority(role="Manager")
    public void doSomeThing(){ }
}
```

不同的地方是，创建重复注解Authority时，加上@Repeatable,指向存储注解Authorities，在使用时候，直接可以重复使用Authority注解。从上面例子看出，java 8里面做法更适合常规的思维，可读性强一点

总结

JEP120没有太多内容，是一个小特性，仅仅是为了提高代码可读性。这次java 8对注解做了2个方面的改进（JEP 104,JEP120），相信注解会比以前使用得更加频繁了。

六、泛型的目标类型推断

> 来源：Java 8新特性探究（六）泛型的目标类型推断

简单理解泛型

泛型是Java SE 1.5的新特性，泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。通俗点将就是“类型的变量”。这种类型变量可以用在类、接口和方法的创建中。

理解Java泛型最简单的方法是把它看成一种便捷语法，能节省你某些Java类型转换(casting)上的操作：

```
List<Apple> box = new ArrayList<Apple>();box.add(new Apple());Apple apple =box.get(0);
```

上面的代码自身已表达的很清楚：box是一个装有Apple对象的List。get方法返回一个Apple对象实例，这个过程不需要进行类型转换。没有泛型，上面的代码需要写成这样：

```
Apple apple = (Apple)box.get(0);
```

泛型的尴尬

泛型的最大优点是提供了程序的类型安全同时可以向后兼容，但也有尴尬的地方，就是每次定义时都要写明泛型的类型，这样显示指定不仅感觉有些冗长，最主要是很多程序员不熟悉泛型，因此很多时候不能够给出正确的类型参数，现在通过编译器自动推断泛型的参数类型，能够减少这样的情况，并提高代码可读性。

java7的泛型类型推断改进

在以前的版本中使用泛型类型，需要在声明并赋值的时候，两侧都加上泛型类型。例如：

```
Map<String, String> myMap = new HashMap<String, String>();
```

你可能觉得:老子在声明变量的的时候已经指明了参数类型，为毛还要在初始化对象时再指定？幸好，在Java SE 7中，这种方式得以改进，现在你可以使用如下语句进行声明并赋值：

```
Map<String, String> myMap = new HashMap<>(); //注意后面的"<>"
```

在这条语句中，编译器会根据变量声明时的泛型类型自动推断出实例化HashMap时的泛型类型。再次提醒一定要注意new HashMap后面的“<>”，只有加上这个“<>”才表示是自动类型推断，否则就是非泛型类型的HashMap，并且在使用编译器编译源代码时会给出一个警告提示。

但是：Java SE 7在创建泛型实例时的类型推断是有限制的：只有构造器的参数化类型在上下文中被显著的声明了，才可以使用类型推断，否则不行。例如：下面的例子在java 7无法正确编译（但现在在java8里面可以编译，因为根据方法参数来自动推断泛型的类型）：

```
List<String> list = new ArrayList<>();
list.add("A");// 由于addAll期望获得Collection<? extends String>类型的参数，因此下面的语句无法通过
list.addAll(new ArrayList<>());
```

Java8的泛型类型推断改进

java8里面泛型的目标类型推断主要2个：

1.支持通过方法上下文推断泛型目标类型

2.支持在方法调用链路当中，泛型类型推断传递到最后一个方法

让我们看看官网的例子

```
class List<E> {
   static <Z> List<Z> nil() { ... };
   static <Z> List<Z> cons(Z head, List<Z> tail) { ... };
   E head() { ... }
}
```

根据JEP101的特性，我们在调用上面方法的时候可以这样写

```
//通过方法赋值的目标参数来自动推断泛型的类型
List<String> l = List.nil();
//而不是显示的指定类型
//List<String> l = List.<String>nil();
//通过前面方法参数类型推断泛型的类型
List.cons(42, List.nil());
//而不是显示的指定类型
//List.cons(42, List.<Integer>nil());
```

总结

以上是JEP101的特性内容了，Java作为静态语言的代表者，可以说类型系统相当丰富。导致类型间互相转换的问题困扰着每个java程序员，通过编译器自动推断类型的东西可以稍微缓解一下类型转换太复杂的问题。 虽然说是小进步，但对于我们天天写代码的程序员，肯定能带来巨大的作用，至少心情更愉悦了~~说不定在java 9里面，我们会得到一个通用的类型var，像js或者scala的一些动态语言那样^_^

七、深入解析日期和时间：JSR310

> 来源：Java 8新特性探究（七）深入解析日期和时间 JSR310

众所周知，日期是商业逻辑计算一个关键的部分，任何企业应用程序都需要处理时间问题。应用程序需要知道当前的时间点和下一个时间点，有时它们还必须计算这两个时间点之间的路径。但java之前的日期做法太令人恶心了，我们先来吐槽一下。

吐槽java.util.Date跟Calendar

Tiago Fernandez做过一次投票，选举最烂的JAVA API，排第一的EJB2.X，第二的就是日期API。

槽点一

最开始的时候，Date既要承载日期信息，又要做日期之间的转换，还要做不同日期格式的显示，职责较繁杂（不懂单一职责，你妈妈知道吗？纯属恶搞~哈哈）

后来从JDK 1.1 开始，这三项职责分开了：

- 使用Calendar类实现日期和时间字段之间转换；
- 使用DateFormat类来格式化和分析日期字符串；
- 而Date只用来承载日期和时间信息。

原有Date中的相应方法已废弃。不过，无论是Date，还是Calendar，都用着太不方便了，这是API没有设计好的地方。

槽点二

坑爹的year和month

```
Date date = new Date(2012,1,1);
System.out.println(date);
输出Thu Feb 01 00:00:00 CST 3912
```

观察输出结果，year是2012+1900，而month，月份参数我不是给了1吗？怎么输出二月（Feb）了？

应该曾有人告诉你，如果你要设置日期，应该使用 java.util.Calendar，像这样…

```
Calendar calendar = Calendar.getInstance();
calendar.set(2013, 8, 2);
```

这样写又不对了，calendar的month也是从0开始的，表达8月份应该用7这个数字，要么就干脆用枚举

```
calendar.set(2013, Calendar.AUGUST, 2);
```

注意上面的代码，Calendar年份的传值不需要减去1900（当然月份的定义和Date还是一样），这种不一致真是让人抓狂！

有些人可能知道，Calendar相关的API是IBM捐出去的，所以才导致不一致。

槽点三

java.util.Date与java.util.Calendar中的所有属性都是可变的

下面的代码，计算两个日期之间的天数....

```
public static void main(String[] args) {
    Calendar birth = Calendar.getInstance();
    birth.set(1975, Calendar.MAY, 26);
    Calendar now = Calendar.getInstance();
    System.out.println(daysBetween(birth, now));
    System.out.println(daysBetween(birth, now)); // 显示 0？
 }

public static long daysBetween(Calendar begin, Calendar end) {
    long daysBetween = 0;
    while(begin.before(end)) {
        begin.add(Calendar.DAY_OF_MONTH, 1);
        daysBetween++;
    }
    return daysBetween;
}
```

daysBetween有点问题，如果连续计算两个Date实例的话，第二次会取得0，因为Calendar状态是可变的，考虑到重复计算的场合，最好复制一个新的Calendar

```
public static long daysBetween(Calendar begin, Calendar end) {
    Calendar calendar = (Calendar) begin.clone(); // 复制
    long daysBetween = 0;
    while(calendar.before(end)) {
        calendar.add(Calendar.DAY_OF_MONTH, 1);
        daysBetween++;
    }
    return daysBetween;
}
```

JSR310

以上种种，导致目前有些第三方的java日期库诞生，比如广泛使用的JODA-TIME，还有Date4j等，虽然第三方库已经足够强大，好用，但还是有兼容问题的，比如标准的JSF日期转换器与joda-time API就不兼容，你需要编写自己的转换器，所以标准的API还是必须的，于是就有了JSR310。

JSR 310实际上有两个日期概念。第一个是Instant，它大致对应于java.util.Date类，因为它代表了一个确定的时间点，即相对于标准Java纪元（1970年1月1日）的偏移量；但与java.util.Date类不同的是其精确到了纳秒级别。

第二个对应于人类自身的观念，比如LocalDate和LocalTime。他们代表了一般的时区概念，要么是日期（不包含时间），要么是时间（不包含日期），类似于java.sql的表示方式。此外，还有一个MonthDay，它可以存储某人的生日（不包含年份）。每个类都在内部存储正确的数据而不是像java.util.Date那样利用午夜12点来区分日期，利用1970-01-01来表示时间。

目前Java8已经实现了JSR310的全部内容。新增了java.time包定义的类表示了日期-时间概念的规则，包括instants, durations, dates, times, time-zones and periods。这些都是基于ISO日历系统，它又是遵循 Gregorian规则的。最重要的一点是值不可变，且线程安全，通过下面一张图，我们快速看下java.time包下的一些主要的类的值的格式，方便理解。

- **LocalDate** **2010-12-03**
- **LocalTime** **11:05:30**
- **LocalDateTime** **2010-12-03T11:05:30**
- **OffsetTime** **11:05:30+01:00**
- **OffsetDateTime** **2010-12-03T11:05:30+01:00**
- **ZonedDateTime** **2010-12-03T11:05:30+01:00 Europe/Paris**

- **Year** **2010**
- **YearMonth** **2010-12**
- **MonthDay** **-12-03**

- **Instant** **2576458258.266 seconds after 1970-01-01**

方法概览

该包的API提供了大量相关的方法，这些方法一般有一致的方法前缀：

of：静态工厂方法。

parse：静态工厂方法，关注于解析。

get：获取某些东西的值。

is：检查某些东西的是否是true。

with：不可变的setter等价物。

plus：加一些量到某个对象。

minus：从某个对象减去一些量。

to：转换到另一个类型。

at：把这个对象与另一个对象组合起来，例如： date.atTime(time)。

与旧的**API**对应关系

Java.time ISO Calendar	Java.util Calendar
Instant	Date
LocalDate, LocalTime, LocalDateTime	Calendar
ZonedDateTime	Calendar
OffsetDateTime, OffsetTime,	Calendar
ZoneId, ZoneOffset, ZoneRules	TimeZone
Week Starts on Monday (1 .. 7) enum MONDAY, TUESDAY, … SUNDAY	Week Starts on Sunday (1 .. 7) int values SUNDAY, MONDAY, … SATURDAY
12 Months (1 .. 12) enum JANUARY, FEBRUARY, …, DECEMBER	12 Months (0 .. 11) int values JANUARY, FEBRUARY, … DECEMBER

简单使用**java.time**的**API**

参考http://jinnianshilongnian.iteye.com/blog/1994164 被我揉在一起，可读性很差，相应的代码都有注释了，我就不过多解释了。

```
public class TimeIntroduction {
    public static void testClock() throws InterruptedException {
        //时钟提供给我们用于访问某个特定 时区的 瞬时时间、日期 和 时间的。
        Clock c1 = Clock.systemUTC(); //系统默认UTC时钟（当前瞬时时间 System.currentTimeMilli
        System.out.println(c1.millis()); //每次调用将返回当前瞬时时间（UTC）
        Clock c2 = Clock.systemDefaultZone(); //系统默认时区时钟（当前瞬时时间）
        Clock c31 = Clock.system(ZoneId.of("Europe/Paris")); //巴黎时区
        System.out.println(c31.millis()); //每次调用将返回当前瞬时时间（UTC）
        Clock c32 = Clock.system(ZoneId.of("Asia/Shanghai"));//上海时区
        System.out.println(c32.millis());//每次调用将返回当前瞬时时间（UTC）
        Clock c4 = Clock.fixed(Instant.now(), ZoneId.of("Asia/Shanghai"));//固定上海时区时钟
        System.out.println(c4.millis());
        Thread.sleep(1000);
        System.out.println(c4.millis()); //不变 即时钟时钟在那一个点不动
        Clock c5 = Clock.offset(c1, Duration.ofSeconds(2)); //相对于系统默认时钟两秒的时钟
        System.out.println(c1.millis());
        System.out.println(c5.millis());
    }
    public static void testInstant() {
        //瞬时时间 相当于以前的System.currentTimeMillis()
        Instant instant1 = Instant.now();
        System.out.println(instant1.getEpochSecond());//精确到秒 得到相对于1970-01-01 00:00:
        System.out.println(instant1.toEpochMilli()); //精确到毫秒
        Clock clock1 = Clock.systemUTC(); //获取系统UTC默认时钟
        Instant instant2 = Instant.now(clock1);//得到时钟的瞬时时间
        System.out.println(instant2.toEpochMilli());
        Clock clock2 = Clock.fixed(instant1, ZoneId.systemDefault()); //固定瞬时时间时钟
        Instant instant3 = Instant.now(clock2);//得到时钟的瞬时时间
        System.out.println(instant3.toEpochMilli());//equals instant1
    }
    public static void testLocalDateTime() {
        //使用默认时区时钟瞬时时间创建 Clock.systemDefaultZone() -->即相对于 ZoneId.systemDefau
        LocalDateTime now = LocalDateTime.now();
        System.out.println(now);
//自定义时区
```

```
        LocalDateTime now2 = LocalDateTime.now(ZoneId.of("Europe/Paris"));
        System.out.println(now2);//会以相应的时区显示日期
//自定义时钟
        Clock clock = Clock.system(ZoneId.of("Asia/Dhaka"));
        LocalDateTime now3 = LocalDateTime.now(clock);
        System.out.println(now3);//会以相应的时区显示日期
//不需要写什么相对时间 如java.util.Date 年是相对于1900 月是从0开始
//2013-12-31 23:59
        LocalDateTime d1 = LocalDateTime.of(2013, 12, 31, 23, 59);
//年月日 时分秒 纳秒
        LocalDateTime d2 = LocalDateTime.of(2013, 12, 31, 23, 59, 59, 11);
//使用瞬时时间 + 时区
        Instant instant = Instant.now();
        LocalDateTime d3 = LocalDateTime.ofInstant(Instant.now(), ZoneId.systemDefault())
        System.out.println(d3);
//解析String--->LocalDateTime
        LocalDateTime d4 = LocalDateTime.parse("2013-12-31T23:59");
        System.out.println(d4);
        LocalDateTime d5 = LocalDateTime.parse("2013-12-31T23:59:59.999");//999毫秒 等价于9
        System.out.println(d5);
//使用DateTimeFormatter API 解析 和 格式化
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
        LocalDateTime d6 = LocalDateTime.parse("2013/12/31 23:59:59", formatter);
        System.out.println(formatter.format(d6));
//时间获取
        System.out.println(d6.getYear());
        System.out.println(d6.getMonth());
        System.out.println(d6.getDayOfYear());
        System.out.println(d6.getDayOfMonth());
        System.out.println(d6.getDayOfWeek());
        System.out.println(d6.getHour());
        System.out.println(d6.getMinute());
        System.out.println(d6.getSecond());
        System.out.println(d6.getNano());
//时间增减
        LocalDateTime d7 = d6.minusDays(1);
        LocalDateTime d8 = d7.plus(1, IsoFields.QUARTER_YEARS);
//LocalDate 即年月日 无时分秒
//LocalTime即时分秒 无年月日
//API和LocalDateTime类似就不演示了
    }
    public static void testZonedDateTime() {
        //即带有时区的date-time 存储纳秒、时区和时差（避免与本地date-time歧义）。
//API和LocalDateTime类似，只是多了时差(如2013-12-20T10:35:50.711+08:00[Asia/Shanghai])
        ZonedDateTime now = ZonedDateTime.now();
        System.out.println(now);
        ZonedDateTime now2 = ZonedDateTime.now(ZoneId.of("Europe/Paris"));
        System.out.println(now2);
//其他的用法也是类似的 就不介绍了
        ZonedDateTime z1 = ZonedDateTime.parse("2013-12-31T23:59:59Z[Europe/Paris]");
        System.out.println(z1);
    }
    public static void testDuration() {
        //表示两个瞬时时间的时间段
        Duration d1 = Duration.between(Instant.ofEpochMilli(System.currentTimeMillis() -
//得到相应的时差
        System.out.println(d1.toDays());
        System.out.println(d1.toHours());
        System.out.println(d1.toMinutes());
        System.out.println(d1.toMillis());
        System.out.println(d1.toNanos());
//1天时差 类似的还有如ofHours()
        Duration d2 = Duration.ofDays(1);
        System.out.println(d2.toDays());
    }
    public static void testChronology() {
        //提供对java.util.Calendar的替换，提供对年历系统的支持
        Chronology c = HijrahChronology.INSTANCE;
        ChronoLocalDateTime d = c.localDateTime(LocalDateTime.now());
        System.out.println(d);
    }
    /**
```

```
     * 新旧日期转换
     */
    public static void testNewOldDateConversion(){
        Instant instant=new Date().toInstant();
        Date date=Date.from(instant);
        System.out.println(instant);
        System.out.println(date);
    }
    public static void main(String[] args) throws InterruptedException {
        testClock();
        testInstant();
        testLocalDateTime();
        testZonedDateTime();
        testDuration();
        testChronology();
        testNewOldDateConversion();
    }
}
```

与**Joda-Time**的区别

其实JSR310的规范领导者Stephen Colebourne，同时也是Joda-Time的创建者，JSR310是在Joda-Time的基础上建立的，参考了绝大部分的API，但并不是说JSR310=JODA-Time，下面几个比较明显的区别是

1. 最明显的变化就是包名（从org.joda.time以及java.time）
2. JSR310不接受NULL值，Joda-Time视NULL值为0
3. JSR310的计算机相关的时间（Instant）和与人类相关的时间（DateTime）之间的差别变得更明显
4. JSR310所有抛出的异常都是DateTimeException的子类。虽然DateTimeException是一个RuntimeException

总结

对比旧的日期API

Java.time	java.util.Calendar以及Date
流畅的API	不流畅的API
实例不可变	实例可变
线程安全	非线程安全

日期与时间处理API，在各种语言中，可能都只是个不起眼的API，如果你没有较复杂的时间处理需求，可能只是利用日期与时间处理API取得系统时间，简单做些显示罢了，然而如果认真看待日期与时间，其复杂程度可能会远超过你的想象，天文、地理、历史、政治、文化等因素，都会影响到你对时间的处理。所以在处理时间上，最好选用JSR310（如果你用java8的话就实现310了），或者Joda-Time。

不止是java面临时间处理的尴尬，其他语言同样也遇到过类似的问题，比如

Arrow：Python 中更好的日期与时间处理库

Moment.js：JavaScript 中的日期库

Noda-Time：.NET 阵营的 Joda-Time 的复制

八、精简的JRE详解

> 来源：Java 8新特性探究（八）精简的JRE详解

Oracle公司如期发布了Java 8正式版！没有让广大javaer失望。对于一个人来说，18岁是人生的转折点，从稚嫩走向成熟，法律意味着你是完全民事行为能力人，不再收益于未成年人保护法，到今年为止，java也走过了18年，java8是一个新的里程碑，带来了前所未有的诸多特性，lambda表达式，Stream API，新的Date time api，多核并发支持，重大安全问题改进等，相信java会越来越好，丰富的类库以及庞大的开源生态环境是其他语言所不具备的，说起丰富的类库，很多同学就吐槽了，java该减肥了，确实是该减肥，java8有个很好的特性，即JEP161(http://openjdk.java.net/jeps/161),该特性定义了Java SE平台规范的一些子集，使java应用程序不需要整个JRE平台即可部署和运行在小型设备上。开发人员可以基于目标硬件的可用资源选择一个合适的JRE运行环境。

好处

1. 更小的Java环境需要更少的计算资源。
2. 一个较小的运行时环境可以更好的优化性能和启动时间。
3. 消除未使用的代码从安全的角度总是好的。
4. 这些打包的应用程序可以下载速度更快。

概念

紧凑的JRE分3种，分别是compact1、compact2、compact3，他们的关系是compact1<compact2<compact3,他们包含的API如下图所示



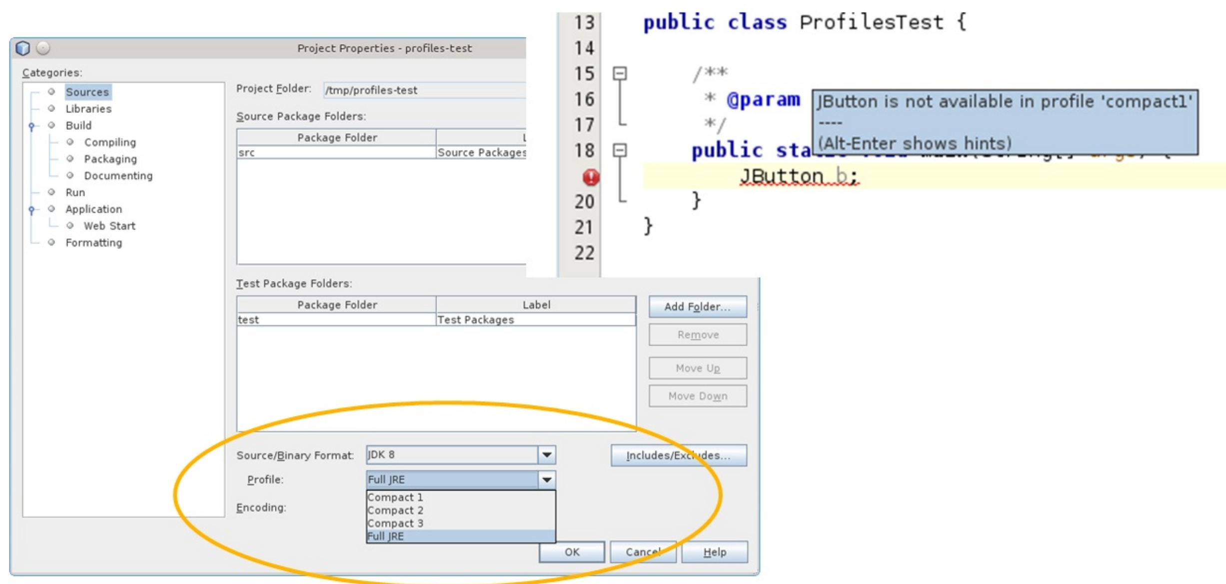
使用javac根据profile编译应用程序

javac –bootclasspath, or javac –profile

如果不符合compact的api，则报错。

```
$ javac -profile compact2 Test.java
Test.java:7: error: ThreadMXBean is not available in profile 'compact2'
 ThreadMXBean bean = ManagementFactory.getThreadMXBean();
 ^
Test.java:7: error: ManagementFactory is not available in profile 'compact2'
 ThreadMXBean bean = ManagementFactory.getThreadMXBean();
                     ^
2 errors
```

使用工具开发的效果



JPEDS工具使用

java8新增一个工具，用来分析应用程序所依赖的profile，有三个参数比较常用 -p，-v，-r

```
import java.util.Set;
import java.util.HashSet;

public class Deps {
  public static void main(String[] args) {
    System.out.println(Math.random());
    Set<String> set = new HashSet<>();
  }
}
```

```
************** PROFILE ********************
jdeps -P Deps.class
Deps.class -> /Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Home/jre/lib/rt.jar
   <unnamed> (Deps.class)
      -> java.io                                            compact1
      -> java.lang                                          compact1
      -> java.util                                          compact1

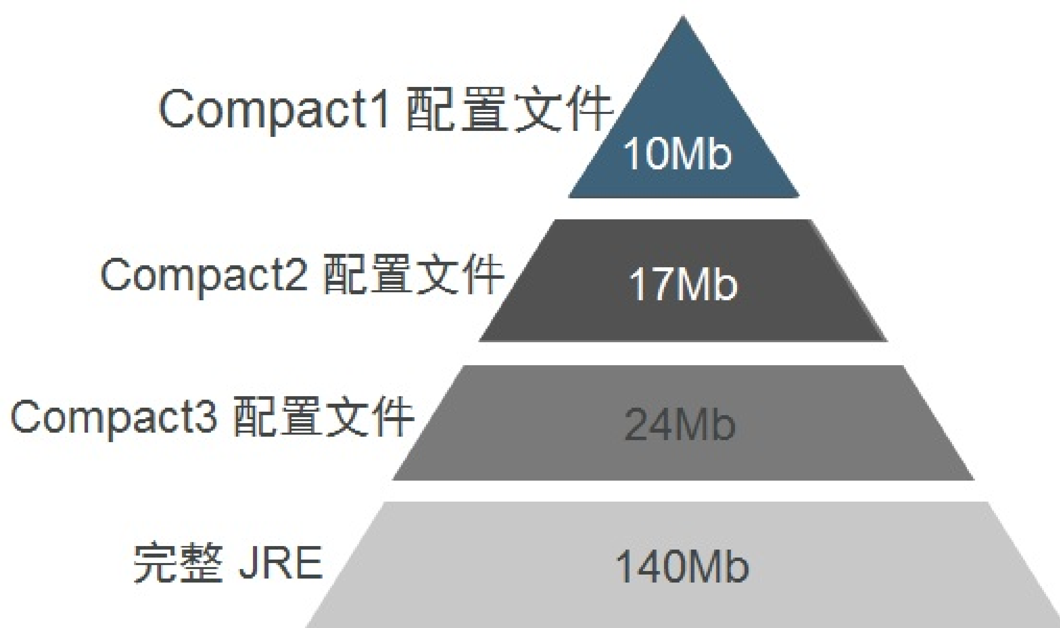
************** VERBOSE ********************
jdeps -v Deps.class
Deps.class -> /Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Home/jre/lib/rt.jar
   Deps (Deps.class)
      -> java.io.PrintStream
      -> java.lang.Math
      -> java.lang.Object
      -> java.lang.String
      -> java.lang.System
      -> java.util.HashSet

************** RECURSIVE ********************
jdeps -R Deps.class
Deps.class -> /Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Home/jre/lib/rt.jar
   <unnamed> (Deps.class)
      -> java.io
      -> java.lang
      -> java.util
/Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Home/jre/lib/jce.jar -> /Library/
   javax.crypto (jce.jar)
      -> java.io
      -> java.lang
      -> java.lang.reflect
      -> java.net
      -> java.nio
      -> java.security
      -> java.security.cert
      -> java.security.spec
      -> java.util
      -> java.util.concurrent
      -> java.util.jar
      -> java.util.regex
      -> java.util.zip
      -> javax.security.auth
      -> sun.security.jca                                   JDK internal API (rt.jar)
      -> sun.security.util                                  JDK internal API (rt.jar)
      -> sun.security.validator                             JDK internal API (rt.jar)
   javax.crypto.interfaces (jce.jar)
      -> java.lang
      -> java.math
      -> java.security
   javax.crypto.spec (jce.jar)
      -> java.lang
      -> java.math
      -> java.security.spec
      -> java.util
/Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Home/jre/lib/rt.jar -> /Library/J
   java.security (rt.jar)
      -> javax.crypto                                       JDK internal API (jce.jar)
   sun.security.util (rt.jar)
      -> javax.crypto                                       JDK internal API (jce.jar)
      -> javax.crypto.interfaces                            JDK internal API (jce.jar)
      -> javax.crypto.spec                                  JDK internal API (jce.jar)
```

在linux上构建profile

```
$ hg clone http://hg.openjdk.java.net/jdk8/jdk8/
$ cd jdk8
$ make images profiles :
## Finished profiles (build time 00:00:27)
----- Build times -------
Start 2013-03-17 14:47:35
End 2013-03-17 14:58:26
00:00:25 corba
00:00:15 demos
00:01:50 hotspot
00:00:24 images
00:00:21 jaxp
00:00:31 jaxws
00:05:37 jdk
00:00:43 langtools
00:00:18 nashorn
00:00:27 profiles
00:10:51 TOTAL
-------------------------
Finished building Java(TM) for target 'images profiles'
$ cd images
$ ls -d *image
j2re-compact1-image j2re-compact2-image j2re-compact3-image j2re-image j2sdk-image
```

编译后**compact**大致的占用空间



总结

如今，物联网正风行一时。我们看到大量不同的设备在市场上出现，每一种的更新速度都越来越快。java需要一个占用资源少的JRE运行环境，紧凑的JRE特性的出现，希望能带来以后的物联网的发展，甚至还是会有大量的java应用程序出现在物联网上面。目前oracle也发布了针对raspberry pi的JRE了。

另外该特性也是为java9的模块化项目做准备，模块化特性是javaer所期待的特性。他是解决业务系统复杂度的一个利器，当然OSGI也是相当的出色。但osgi对于新学者来说未免太复杂了。

九、跟OOM：Permgen说再见吧

> 来源：Java 8新特性探究（九）跟OOM：Permgen说再见吧

很多开发者都在其系统中见过“java.lang.OutOfMemoryError: PermGen space”这一问题。这往往是由类加载器相关的内存泄漏以及新类加载器的创建导致的，通常出现于代码热部署时。相对于正式产品，该问题在开发机上出现的频率更高，在产品中最常见的“问题”是默认值太低了。常用的解决方法是将其设置为256MB或更高。

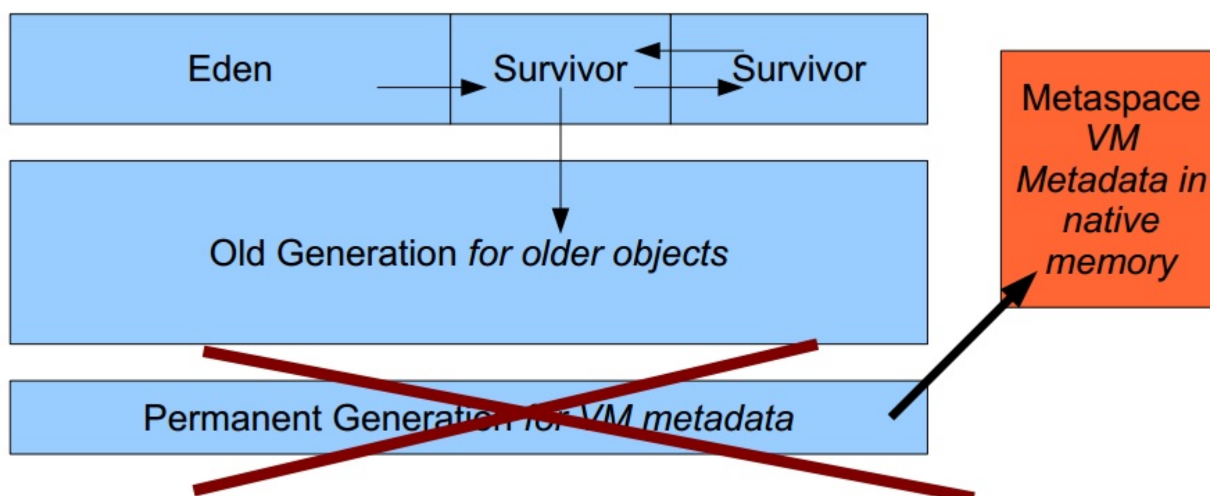
PermGen space简单介绍

PermGen space的全称是Permanent Generation space,是指内存的永久保存区域，说说为什么会内存益出：这一部分用于存放Class和Meta的信息,Class在被 Load的时候被放入PermGen space区域，它和和存放Instance的Heap区域不同,所以如果你的APP会LOAD很多CLASS的话,就很可能出现PermGen space错误。这种错误常见在web服务器对JSP进行pre compile的时候。

JVM 种类有很多，比如 Oralce-Sun Hotspot, Oralce JRockit, IBM J9, Taobao JVM（淘宝好样的！）等等。当然武林盟主是Hotspot了，这个毫无争议。需要注意的是，PermGen space是Oracle-Sun Hotspot才有，JRockit以及J9是没有这个区域。

元空间（MetaSpace）一种新的内存空间诞生

JDK8 HotSpot JVM 将移除永久区，使用本地内存来存储类元数据信息并称之为：元空间（Metaspace）；这与Oracle JRockit 和IBM JVM's很相似，如下图所示



这意味着不会再有java.lang.OutOfMemoryError: PermGen问题，也不再需要你进行调优及监控内存空间的使用……但请等等，这么说还为时过早。在默认情况下，这些改变是透明的，接下来我们的展示将使你知道仍然要关注类元数据内存的占用。请一定要牢记，这个新特性也不能神奇地消除类和类加载器导致的内存泄漏。

java8中metaspace总结如下：

PermGen 空间的状况

这部分内存空间将全部移除。

JVM的参数：PermSize 和 MaxPermSize 会被忽略并给出警告（如果在启用时设置了这两个参数）。

Metaspace 内存分配模型

大部分类元数据都在本地内存中分配。

用于描述类元数据的“klasses”已经被移除。

Metaspace 容量

默认情况下，类元数据只受可用的本地内存限制（容量取决于是32位或是64位操作系统的可用虚拟内存大小）。

新参数（MaxMetaspaceSize）用于限制本地内存分配给类元数据的大小。如果没有指定这个参数，元空间会在运行时根据需要动态调整。

Metaspace 垃圾回收

对于僵死的类及类加载器的垃圾回收将在元数据使用达到“MaxMetaspaceSize”参数的设定值时进行。

适时地监控和调整元空间对于减小垃圾回收频率和减少延时是很有必要的。持续的元空间垃圾回收说明，可能存在类、类加载器导致的内存泄漏或是大小设置不合适。

Java 堆内存的影响

一些杂项数据已经移到Java堆空间中。升级到JDK8之后，会发现Java堆 空间有所增长。

Metaspace 监控

元空间的使用情况可以从HotSpot1.8的详细GC日志输出中得到。

Jstat 和 JVisualVM两个工具，在使用b75版本进行测试时，已经更新了，但是还是能看到老的PermGen空间的出现。

前面已经从理论上充分说明，下面让我们通过“泄漏”程序进行新内存空间的观察……

PermGen vs. Metaspace 运行时比较

为了更好地理解Metaspace内存空间的运行时行为，

将进行以下几种场景的测试：

1. 使用JDK1.7运行Java程序，监控并耗尽默认设定的85MB大小的PermGen内存空间。
2. 使用JDK1.8运行Java程序，监控新Metaspace内存空间的动态增长和垃圾回收过程。
3. 使用JDK1.8运行Java程序，模拟耗尽通过“MaxMetaspaceSize”参数设定的128MB大小的Metaspace内存空间。

首先建立了一个模拟PermGen OOM的代码

```
public class ClassA {
 public void method(String name) {
  // do nothing
 }
}
```

上面是一个简单的ClassA，把他编译成class字节码放到D：/classes下面，测试代码中用URLClassLoader来加载此类型上面类编译成class

```
/**
 * 模拟PermGen OOM
 * @author benhail
 */
public class OOMTest {
    public static void main(String[] args) {
        try {
            //准备url
            URL url = new File("D:/classes").toURI().toURL();
            URL[] urls = {url};
            //获取有关类型加载的JMX接口
            ClassLoadingMXBean loadingBean = ManagementFactory.getClassLoadingMXBean();
            //用于缓存类加载器
            List<ClassLoader> classLoaders = new ArrayList<ClassLoader>();
            while (true) {
                //加载类型并缓存类加载器实例
                ClassLoader classLoader = new URLClassLoader(urls);
                classLoaders.add(classLoader);
                classLoader.loadClass("ClassA");
                //显示数量信息（共加载过的类型数目，当前还有效的类型数目，已经被卸载的类型数目）
                System.out.println("total: " + loadingBean.getTotalLoadedClassCount());
                System.out.println("active: " + loadingBean.getLoadedClassCount());
                System.out.println("unloaded: " + loadingBean.getUnloadedClassCount());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

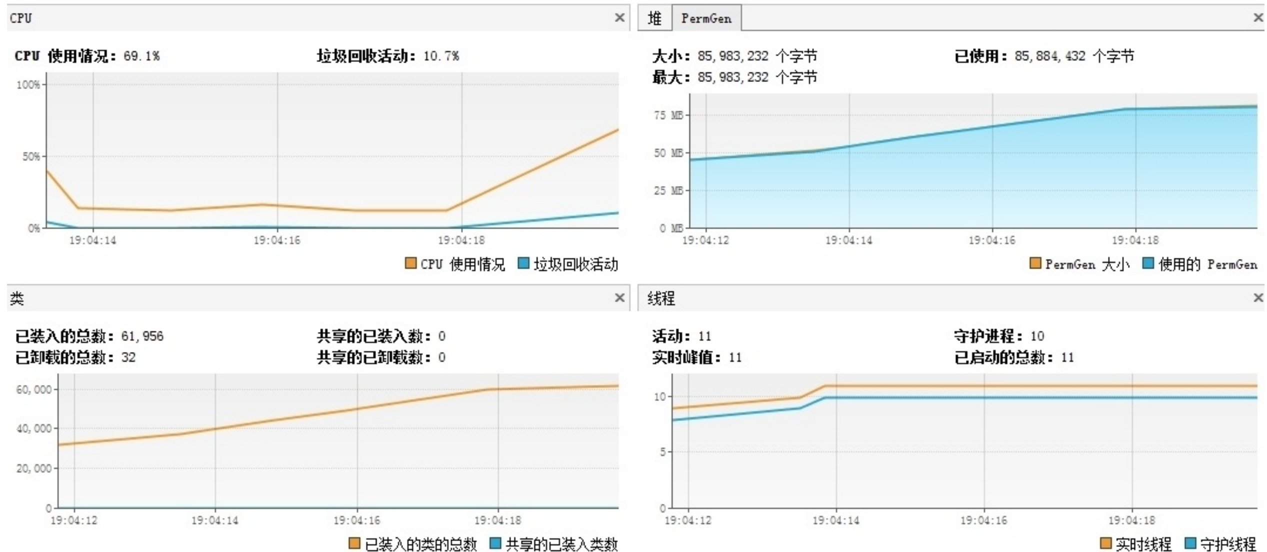
虚拟机器参数设置如下：-verbose -verbose:gc

设置-verbose参数是为了获取类型加载和卸载的信息

设置-verbose:gc是为了获取垃圾收集的相关信息

JDK 1.7 @64-bit – PermGen 耗尽测试

Java1.7的PermGen默认空间为85 MB（或者可以通过-XX:MaxPermSize=XXXm指定）



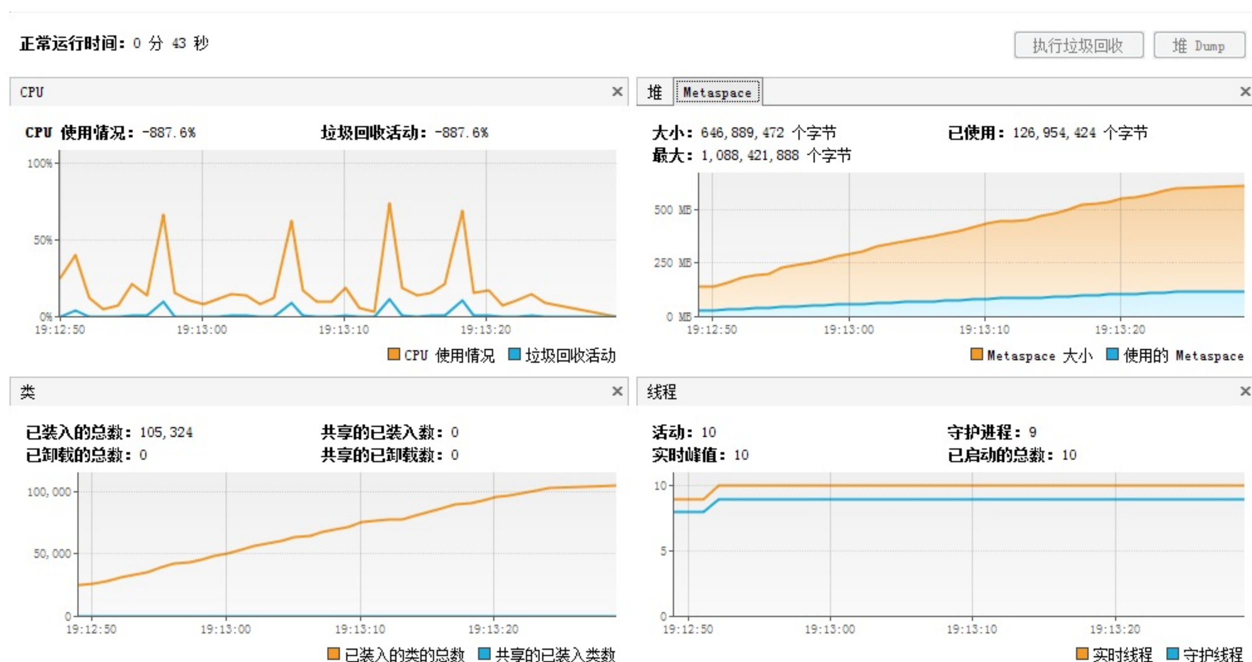
可以从上面的JVisualVM的截图看出：当加载超过6万个类之后，PermGen被耗尽。我们也能通过程序和GC的输出观察耗尽的过程。

程序输出(摘取了部分)

```
......
[Loaded ClassA from file:/D:/classes/]
total: 64887
active: 64887
unloaded: 0
[GC 245041K->213978K(536768K), 0.0597188 secs]
[Full GC 213978K->211425K(644992K), 0.6456638 secs]
[GC 211425K->211425K(656448K), 0.0086696 secs]
[Full GC 211425K->211411K(731008K), 0.6924754 secs]
[GC 211411K->211411K(726528K), 0.0088992 secs]
...............
java.lang.OutOfMemoryError: PermGen space
```

JDK 1.8 @64-bit – Metaspace大小动态调整测试

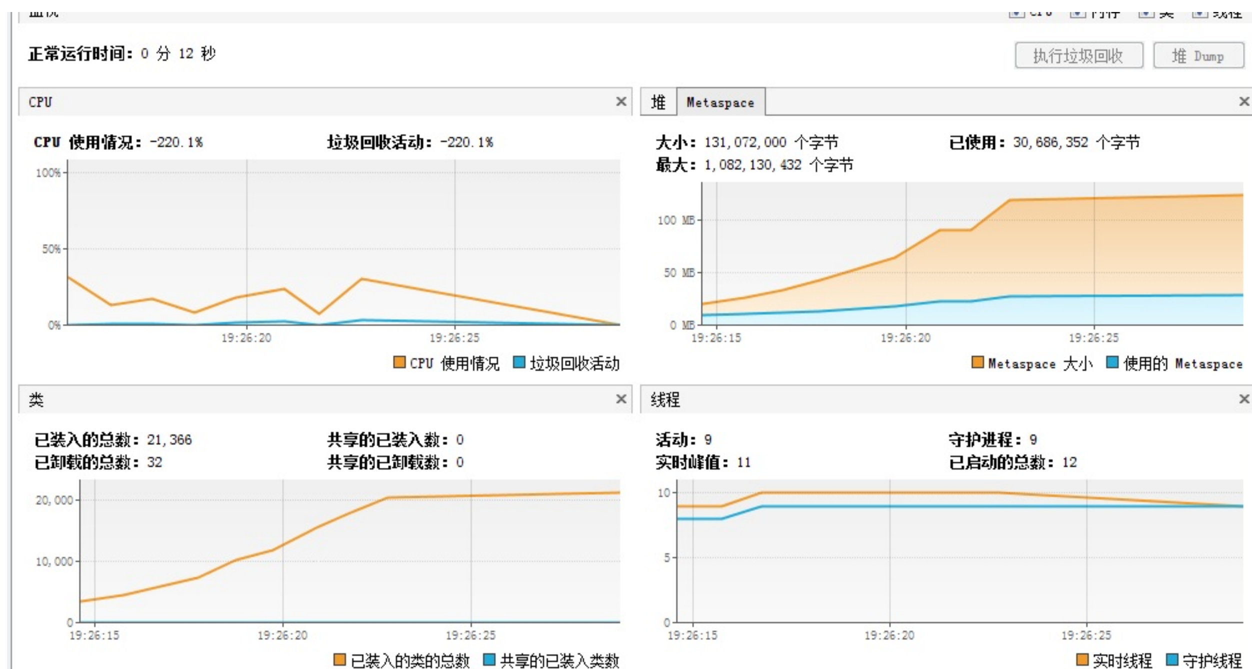
Java的Metaspace空间：不受限制 （默认）



从上面的截图可以看到，JVM Metaspace进行了动态扩展，本地内存的使用由20MB增长到646MB，以满足程序中不断增长的类数据内存占用需求。我们也能观察到JVM的垃圾回收事件—试图销毁僵死的类或类加载器对象。但是，由于我们程序的泄漏，JVM别无选择只能动态扩展Metaspace内存空间。程序加载超过10万个类，而没有出现OOM事件。

JDK 1.8 @64-bit – Metaspace 受限测试

Java的Metaspace空间：128MB（-XX:MaxMetaspaceSize=128m）



可以从上面的JVisualVM的截图看出：当加载超过2万个类之后，Metaspace被耗尽；与JDK1.7运行时非常相似。我们也能通过程序和GC的输出观察耗尽的过程。另一个有趣的现象是，保留的原生内存占用量是设定的最大大小两倍之多。这可能表明，如果可能的话，可微调元空间容量大小策略，来避免本地内存的浪费。

从Java程序的输出中看到如下异常。

```
[Loaded ClassA from file:/D:/classes/]
total: 21393
active: 21393
unloaded: 0
[GC (Metadata GC Threshold) 64306K->57010K(111616K), 0.0145502 secs]
[Full GC (Metadata GC Threshold) 57010K->56810K(122368K), 0.1068084 secs]
java.lang.OutOfMemoryError: Metaspace
```

在设置了MaxMetaspaceSize的情况下，该空间的内存仍然会耗尽，进而引发“java.lang.OutOfMemoryError: Metadata space”错误。因为类加载器的泄漏仍然存在，而通常Java又不希望无限制地消耗本机内存，因此设置一个类似于MaxPermSize的限制看起来也是合理的。

总结

1. 之前不管是不是需要，JVM都会吃掉那块空间……如果设置得太小，JVM会死掉；如果设置得太大，这块内存就被JVM浪费了。理论上说，现在你完全可以不关注这个，因为JVM会在运行时自动调校为“合适的大小”；
2. 提高Full GC的性能，在Full GC期间，Metadata到Metadata pointers之间不需要扫描了，别小看这几纳秒时间；
3. 隐患就是如果程序存在内存泄露，像OOMTest那样，不停的扩展metaspace的空间，会导致机器的内存不足，所以还是要有必要的调试和监控。

十、StampedLock将是解决同步问题的新宠

> 来源：Java 8新特性探究（十）StampedLock将是解决同步问题的新宠

Java8就像一个宝藏，一个小的API改进，也足与写一篇文章，比如同步，一直是多线程并发编程的一个老话题，相信没有人喜欢同步的代码，这会降低应用的吞吐量等性能指标，最坏的时候会挂起死机，但是即使这样你也没得选择，因为要保证信息的正确性。所以本文决定将从synchronized、Lock到Java8新增的StampedLock进行对比分析，相信StampedLock不会让大家失望。

synchronized

在java5之前，实现同步主要是使用synchronized。它是Java语言的关键字，当它用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码。

有四种不同的同步块：

1. 实例方法
2. 静态方法
3. 实例方法中的同步块
4. 静态方法中的同步块

大家对此应该不陌生，所以不多讲了，以下是代码示例

```
synchronized(this)
// do operation
}
```

小结：在多线程并发编程中Synchronized一直是元老级角色，很多人都会称呼它为重量级锁，但是随着Java SE1.6对Synchronized进行了各种优化之后，性能上也有所提升。

Lock

```
rwlock.writeLock().lock();
try {
// do operation
} finally {
rwlock.writeLock().unlock();
}
```

它是Java 5在java.util.concurrent.locks新增的一个API。

Lock是一个接口，核心方法是lock()，unlock()，tryLock()，实现类有ReentrantLock, ReentrantReadWriteLock.ReadLock, ReentrantReadWriteLock.WriteLock；

ReentrantReadWriteLock, ReentrantLock 和synchronized锁都有相同的内存语义。

与synchronized不同的是，Lock完全用Java写成，在java这个层面是无关JVM实现的。Lock提供更灵活的锁机制，很多synchronized 没有提供的许多特性，比如锁投票，定时锁等候和中断锁等候，但因为lock是通过代码实现的，要保证锁定一定会被释放，就必须将unLock()放到finally{}中

下面是Lock的一个代码示例

```
class Point {
   private double x, y;
   private final StampedLock sl = new StampedLock();
   void move(double deltaX, double deltaY) { // an exclusively locked method
     long stamp = sl.writeLock();
     try {
       x += deltaX;
       y += deltaY;
     } finally {
       sl.unlockWrite(stamp);
     }
   }
  //下面看看乐观读锁案例
   double distanceFromOrigin() { // A read-only method
     long stamp = sl.tryOptimisticRead(); //获得一个乐观读锁
     double currentX = x, currentY = y; //将两个字段读入本地局部变量
     if (!sl.validate(stamp)) { //检查发出乐观读锁后同时是否有其他写锁发生？
        stamp = sl.readLock(); //如果没有，我们再次获得一个读悲观锁
        try {
          currentX = x; // 将两个字段读入本地局部变量
          currentY = y; // 将两个字段读入本地局部变量
        } finally {
           sl.unlockRead(stamp);
        }
     }
     return Math.sqrt(currentX * currentX + currentY * currentY);
   }
//下面是悲观读锁案例
   void moveIfAtOrigin(double newX, double newY) { // upgrade
     // Could instead start with optimistic, not read mode
     long stamp = sl.readLock();
     try {
       while (x == 0.0 && y == 0.0) { //循环，检查当前状态是否符合
         long ws = sl.tryConvertToWriteLock(stamp); //将读锁转为写锁
         if (ws != 0L) { //这是确认转为写锁是否成功
           stamp = ws; //如果成功 替换票据
           x = newX; //进行状态改变
           y = newY; //进行状态改变
           break;
         }
         else { //如果不能成功转换为写锁
           sl.unlockRead(stamp); //我们显式释放读锁
           stamp = sl.writeLock(); //显式直接进行写锁 然后再通过循环再试
         }
       }
     } finally {
       sl.unlock(stamp); //释放读锁或写锁
     }
   }
 }
```

小结：比synchronized更灵活、更具可伸缩性的锁定机制，但不管怎么说还是synchronized代码要更容易书写些

StampedLock

它是java8在java.util.concurrent.locks新增的一个API。

ReentrantReadWriteLock 在沒有任何读写锁时，才可以取得写入锁，这可用于实现了悲观读取（Pessimistic Reading），即如果执行中进行读取时，经常可能有另一执行要写入的需求，为了保持同步，ReentrantReadWriteLock 的读取锁定就可派上用场。

然而，如果读取执行情况很多，写入很少的情况下，使用 ReentrantReadWriteLock 可能会使写入线程遭遇饥饿（Starvation）问题，也就是写入线程吃吃无法竞争到锁定而一直处于等待状态。

StampedLock控制锁有三种模式（写，读，乐观读），一个StampedLock状态是由版本和模式两个部分组成，锁获取方法返回一个数字作为票据stamp，它用相应的锁状态表示并控制访问，数字0表示没有写锁被授权访问。在读锁上分为悲观锁和乐观锁。

所谓的乐观读模式，也就是若读的操作很多，写的操作很少的情况下，你可以乐观地认为，写入与读取同时发生几率很少，因此不悲观地使用完全的读取锁定，程序可以查看读取资料之后，是否遭到写入执行的变更，再采取后续的措施（重新读取变更信息，或者抛出异常），这一个小小改进，可大幅度提高程序的吞吐量！！

下面是java doc提供的StampedLock一个例子

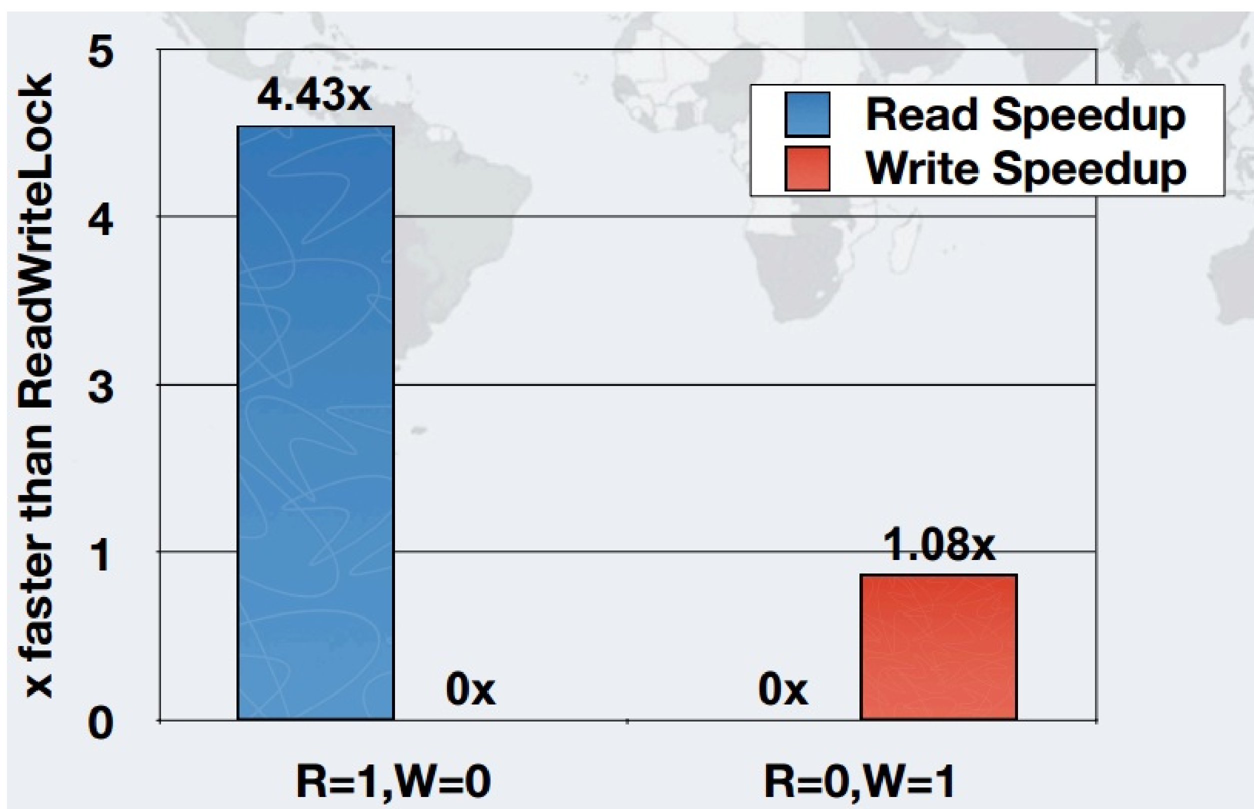
```
class Point {
   private double x, y;
   private final StampedLock sl = new StampedLock();
   void move(double deltaX, double deltaY) { // an exclusively locked method
     long stamp = sl.writeLock();
     try {
       x += deltaX;
       y += deltaY;
     } finally {
       sl.unlockWrite(stamp);
     }
   }
  //下面看看乐观读锁案例
   double distanceFromOrigin() { // A read-only method
     long stamp = sl.tryOptimisticRead(); //获得一个乐观读锁
     double currentX = x, currentY = y; //将两个字段读入本地局部变量
     if (!sl.validate(stamp)) { //检查发出乐观读锁后同时是否有其他写锁发生？
        stamp = sl.readLock(); //如果没有，我们再次获得一个读悲观锁
        try {
          currentX = x; // 将两个字段读入本地局部变量
          currentY = y; // 将两个字段读入本地局部变量
        } finally {
           sl.unlockRead(stamp);
        }
     }
     return Math.sqrt(currentX * currentX + currentY * currentY);
   }
//下面是悲观读锁案例
   void moveIfAtOrigin(double newX, double newY) { // upgrade
     // Could instead start with optimistic, not read mode
     long stamp = sl.readLock();
     try {
       while (x == 0.0 && y == 0.0) { //循环，检查当前状态是否符合
         long ws = sl.tryConvertToWriteLock(stamp); //将读锁转为写锁
         if (ws != 0L) { //这是确认转为写锁是否成功
           stamp = ws; //如果成功 替换票据
           x = newX; //进行状态改变
           y = newY; //进行状态改变
           break;
         }
         else { //如果不能成功转换为写锁
           sl.unlockRead(stamp); //我们显式释放读锁
           stamp = sl.writeLock(); //显式直接进行写锁 然后再通过循环再试
         }
       }
     } finally {
       sl.unlock(stamp); //释放读锁或写锁
     }
   }
 }
```

小结：

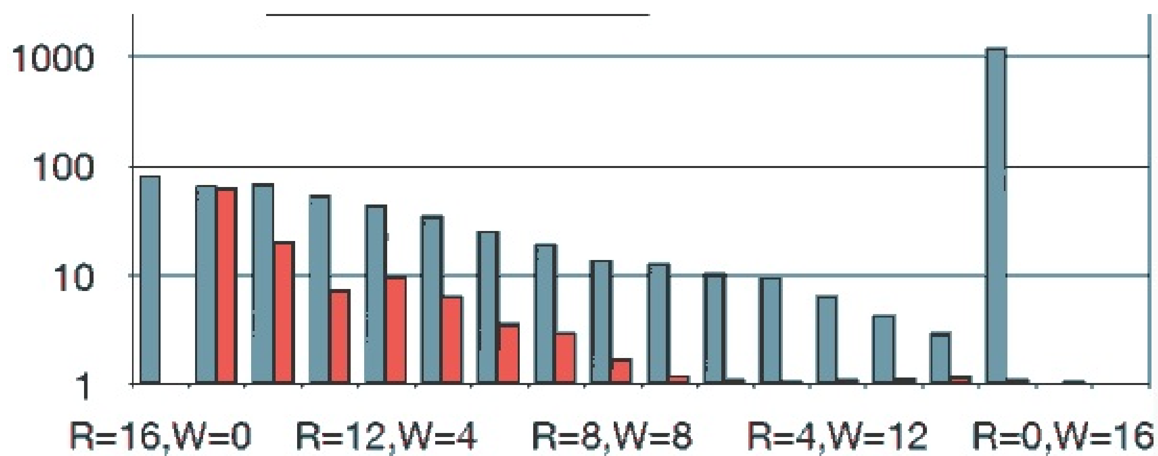
StampedLock要比ReentrantReadWriteLock更加廉价，也就是消耗比较小。

StampedLock与ReadWriteLock性能对比

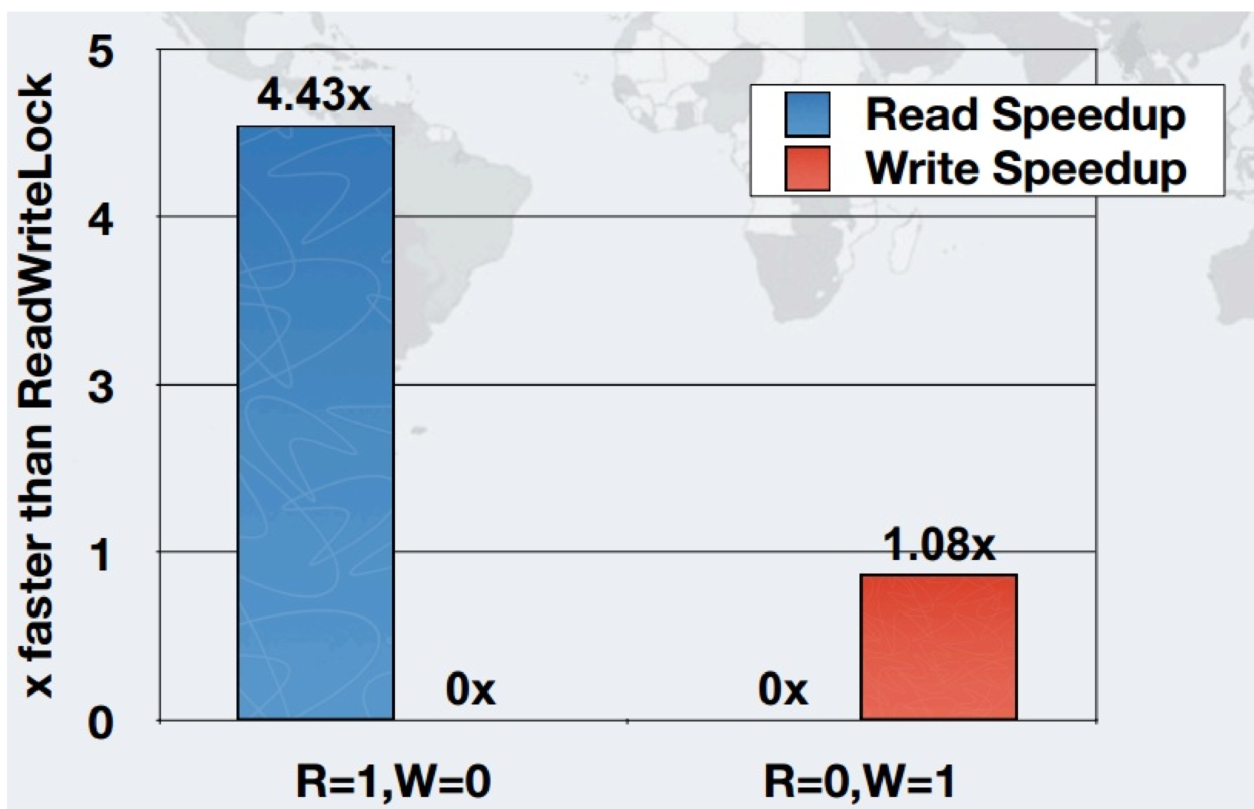
下图是和ReadWritLock相比，在一个线程情况下，是读速度其4倍左右，写是1倍。



下图是六个线程情况下，读性能是其几十倍，写性能也是近10倍左右：



下图是吞吐量提高：



总结

1. synchronized是在JVM层面上实现的，不但可以通过一些监控工具监控synchronized的锁定，而且在代码执行时出现异常，JVM会自动释放锁定；
2. ReentrantLock、ReentrantReadWriteLock,、StampedLock都是对象层面的锁定，要保证锁定一定会被释放，就必须将unLock()放到finally{}中；
3. StampedLock 对吞吐量有巨大的改进，特别是在读线程越来越多的场景下；
4. StampedLock有一个复杂的API，对于加锁操作，很容易误用其他方法;
5. 当只有少量竞争者的时候，synchronized是一个很好的通用的锁实现;
6. 当线程增长能够预估，ReentrantLock是一个很好的通用的锁实现;

StampedLock 可以说是Lock的一个很好的补充，吞吐量以及性能上的提升足以打动很多人了，但并不是说要替代之前Lock的东西，毕竟他还是有些应用场景的，起码API比StampedLock容易入手，下篇博文争取更新快一点，可能会是Nashorn的内容，这里允许我先卖个关子。。。

十一：Base64详解

> 来源：Java 8新特性探究（十一）Base64详解

BASE64 编码是一种常用的字符编码，在很多地方都会用到。但base64不是安全领域下的加密解密算法。能起到安全作用的效果很差，而且很容易破解，他核心作用应该是传输数据的正确性，有些网关或系统只能使用ASCII字符。Base64就是用来将非ASCII字符的数据转换成ASCII字符的一种方法，而且base64特别适合在http，mime协议下快速传输数据。

JDK里面实现Base64的API

在JDK1.6之前，JDK核心类一直没有Base64的实现类，有人建议用Sun/Oracle JDK里面的sun.misc.BASE64Encoder 和 sun.misc.BASE64Decoder，使用它们的优点就是不需要依赖第三方类库，缺点就是可能在未来版本会被删除（用maven编译会发出警告），而且性能不佳，后面会有性能测试。

JDK1.6中添加了另一个Base64的实现，javax.xml.bind.DatatypeConverter两个静态方法parseBase64Binary 和 printBase64Binary，隐藏在javax.xml.bind包下面，不被很多开发者知道。

在Java 8在java.util包下面实现了BASE64编解码API，而且性能不俗，API也简单易懂，下面展示下这个类的使用例子。

java.util.Base64

该类提供了一套静态方法获取下面三种BASE64编解码器：

1）Basic编码：是标准的BASE64编码，用于处理常规的需求

```
// 编码
String asB64 = Base64.getEncoder().encodeToString("some string".getBytes("utf-8"));
System.out.println(asB64); // 输出为: c29tZSBzdHJpbmc=
// 解码
byte[] asBytes = Base64.getDecoder().decode("c29tZSBzdHJpbmc=");
System.out.println(new String(asBytes, "utf-8")); // 输出为: some string
```

2）URL编码：使用下划线替换URL里面的反斜线“/”

```
String urlEncoded = Base64.getUrlEncoder().encodeToString("subjects?abcd".getBytes("utf-8
System.out.println("Using URL Alphabet: " + urlEncoded);
// 输出为:
Using URL Alphabet: c3ViamVjdHM_YWJjZA==
```

3）MIME编码：使用基本的字母数字产生BASE64输出，而且对MIME格式友好：每一行输出不超过76个字符，而且每行以“\r\n”符结束。

```
StringBuilder sb = new StringBuilder();
for (int t = 0; t < 10; ++t) {
  sb.append(UUID.randomUUID().toString());
}
byte[] toEncode = sb.toString().getBytes("utf-8");
String mimeEncoded = Base64.getMimeEncoder().encodeToString(toEncode);
System.out.println(mimeEncoded);
```

第三方实现Base64的API

首先便是常用的Apache Commons Codec library里面的org.apache.commons.codec.binary.Base64；

第二个便是Google Guava库里面的com.google.common.io.BaseEncoding.base64() 这个静态方法；

第三个是net.iharder.Base64，这个jar包就一个类；

最后一个，号称Base64编码速度最快的MigBase64，而且是10年前的实现，到现在是否能保持这个称号，测一测便知道；

Base64编码性能测试

上面讲了一共7种实现Base64编码，Jdk里面3种，第三方实现4种，一旦有选择，则有必要将他们进行一次高低对比，性能测试是最直接的方式

首先来定义两个接口

```
private static interface Base64Codec
    {
        public String encode(final byte[] data);
        public byte[] decode(final String base64) throws IOException;
    }
    private static interface Base64ByteCodec
    {
        public byte[] encodeBytes(final byte[] data);
        public byte[] decodeBytes(final byte[] base64) throws IOException;
    }
```

两个接口区别就是其中一个接口方法参数接收byte数组，返回byte数组，因为byte->byte相比String->byte或者byte->String性能上会快一点，所以区分两组来测试

```
private static final Base64Codec[] m_codecs = { new GuavaImpl(), new JavaXmlImpl(),
        new Java8Impl(), new SunImpl(), new ApacheImpl(),new MiGBase64Impl(),new IHarderI
private static final Base64ByteCodec[] m_byteCodecs = {
        new ApacheImpl(), new Java8Impl(),new MiGBase64Impl(),new IHarderImpl() };
```

从上面看出，其中支持byte->byte只有4中API；

7个Base64的实现类

```
private static class Java8Impl implements Base64Codec, Base64ByteCodec
    {
        private final Base64.Decoder m_decoder = Base64.getDecoder();
        private final Base64.Encoder m_encoder = Base64.getEncoder();
        @Override
        public String encode(byte[] data) {
            return m_encoder.encodeToString(data);
        }
        @Override
        public byte[] decode(String base64) throws IOException {
            return m_decoder.decode(base64);
        }
        public byte[] encodeBytes(byte[] data) {
            return m_encoder.encode( data );
        }
        public byte[] decodeBytes(byte[] base64) throws IOException {
            return m_decoder.decode( base64 );
        }
    }
    private static class JavaXmlImpl implements Base64Codec //no byte[] implementation
    {
        public String encode(byte[] data) {
            return DatatypeConverter.printBase64Binary( data );
        }
        public byte[] decode(String base64) throws IOException {
            return DatatypeConverter.parseBase64Binary( base64 );
        }
    }
..............
```

后面代码基本就是各种API实现Base64的代码了，就不详细列出。

主要测试手段是，生成100M的随机数，分成100byte或者1000byte的块，然后将他们分别编码和解码，记录时间，如下方法

```
private static TestResult testByteCodec( final Base64ByteCodec codec, final List<byte[]>
        final List<byte[]> encoded = new ArrayList<byte[]>( buffers.size() );
        final long start = System.currentTimeMillis();
        for ( final byte[] buf : buffers )
            encoded.add( codec.encodeBytes(buf) );
        final long encodeTime = System.currentTimeMillis() - start;
        final List<byte[]> result = new ArrayList<byte[]>( buffers.size() );
        final long start2 = System.currentTimeMillis();
        for ( final byte[] ar : encoded )
            result.add( codec.decodeBytes(ar) );
        final long decodeTime = System.currentTimeMillis() - start2;
        for ( int i = 0; i < buffers.size(); ++i )
        {
            if ( !Arrays.equals( buffers.get( i ), result.get( i ) ) )
                System.out.println( "Diff at pos = " + i );
        }
        return new TestResult( encodeTime / 1000.0, decodeTime / 1000.0 );
    }
```

测试结果

jvm参数：-Xms512m -Xmx4G

byte->byte

Name	Encode, 100 bytes	Decode, 100 bytes	Encode, 1000 bytes	Decode, 1000 bytes	Encode, 100000000 bytes	Decode, 100000000 bytes
ApacheImpl	1.492 sec	1.424 sec	0.84 sec	0.781 sec	0.828 sec	0.791 sec
IHarderImpl	0.292 sec	0.658 sec	0.26 sec	0.624 sec	0.243 sec	0.625 sec
Java8Impl	0.182 sec	0.281 sec	0.152 sec	0.251 sec	0.138 sec	0.241 sec
MiGBase64Impl	0.233 sec	0.545 sec	0.203 sec	0.483 sec	0.189 sec	0.479 sec

byte->String

Name	Encode, 100 bytes	Decode, 100 bytes	Encode, 1000 bytes	Decode, 1000 bytes	Encode, 100000000 bytes	Decode, 100000000 bytes
ApacheImpl	1.683 sec	1.691 sec	0.967 sec	1.029 sec	0.912 sec	0.948 sec
GuavaImpl	1.03 sec	1.599 sec	0.991 sec	1.425 sec	0.977 sec	1.445 sec
IHarderImpl	0.434 sec	0.891 sec	0.385 sec	0.807 sec	0.361 sec	0.793 sec
Java8Impl	0.274 sec	0.347 sec	0.253 sec	0.295 sec	0.234 sec	0.282 sec
JavaXmlImpl	0.243 sec	0.376 sec	0.241 sec	0.339 sec	0.249 sec	0.316 sec
MiGBase64Impl	0.252 sec	0.67 sec	0.243 sec	0.596 sec	0.251 sec	0.58 sec
SunImpl	3.613 sec	2.796 sec	1.418 sec	1.566 sec	1.117 sec	1.581 sec

一切都很明显了，从上面看出，sun的表现不是很好，IHarder和MigBase64性能可以接受，传说MigBase64性能第一，那也是过去了，在这次测试结果中，新的java8 base64运行速度最好，javaXml表现次之。

总结

如果你需要一个性能好，可靠的Base64编解码器，不要找JDK外面的了，java8里面的java.util.Base64以及java6中隐藏很深的javax.xml.bind.DatatypeConverter,他们两个都是不错的选择。

本篇中所有代码都在http://git.oschina.net/benhail/javase8-sample ，欢迎大家去关注下载

十二、Nashorn ：新犀牛

> 来源：Java 8新特性探究（十二）Nashorn ：新犀牛

Nashorn是什么

Nashorn，发音“nass-horn”,是德国二战时一个坦克的命名，同时也是java8新一代的javascript引擎–替代老旧，缓慢的Rhino，符合 ECMAScript-262 5.1 版语言规范。你可能想javascript是运行在web浏览器，提供对html各种dom操作，但是Nashorn不支持浏览器DOM的对象。这个需要注意的一个点。

关于Nashorn的入门

主要是两个方面，jjs工具以及javax.script包下面的API：

jjs是在java_home/bin下面自带的，作为例子，让我们创建一个func.js， 内容如下：

```
function f(){return 1;};
print(f() + 1);
```

运行这个文件，把这个文件作为参数传给jjs

```
jjs func.js
```

输出结果：2

另一个方面是javax.script，也是以前Rhino余留下来的API

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName( "JavaScript" );
System.out.println( engine.getClass().getName() );
System.out.println( "Result:" + engine.eval( "function f() { return 1; }; f() + 1;" ) );
```

输出如下：

jdk.nashorn.api.scripting.NashornScriptEngine

Result: 2

基本用法也可以去http://my.oschina.net/jsmagic/blog/212455 这篇博文参考一下；

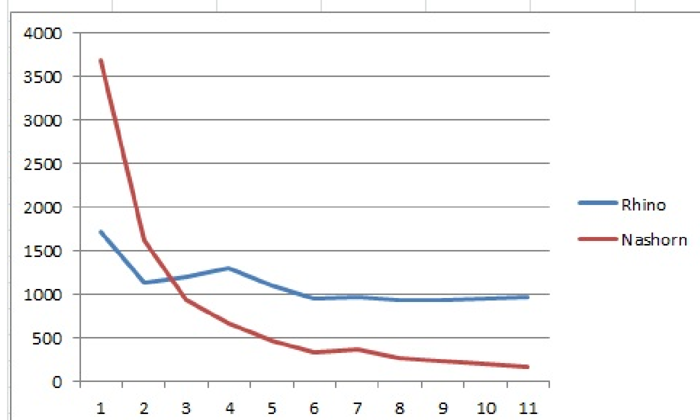
Nashorn VS Rhino

javascript运行在jvm已经不是新鲜事了，Rhino早在jdk6的时候已经存在，但现在为何要替代Rhino，官方的解释是Rhino相比其他javascript引擎（比如google的V8）实在太慢了，要改造Rhino还不如重写。既然性能是Nashorn的一个亮点，下面就测试下性能对比，为了对比两者之间的性能，需要用到Esprima，一个ECMAScript解析框架，用它来解析未压缩版的jquery（大约268kb），测试核心代码如下：

```
static void rhino(String parser, String code) {
        String source = "speedtest";
        int line = 1;
        Context context = Context.enter();
        context.setOptimizationLevel(9);
        try {
            Scriptable scope = context.initStandardObjects();
            context.evaluateString(scope, parser, source, line, null);
            ScriptableObject.putProperty(scope, "$code", Context.javaToJS(code, scope));
            Object tree = new Object();
            Object tokens = new Object();
            for (int i = 0; i < RUNS; ++i) {
                long start = System.nanoTime();
                tree = context.evaluateString(scope, "esprima.parse($code)", source, line
                tokens = context.evaluateString(scope, "esprima.tokenize($code)", source,
                long stop = System.nanoTime();
                System.out.println("Run #" + (i + 1) + ": " + Math.round((stop - start) /
            }
        } finally {
            Context.exit();
            System.gc();
        }
    }
    static void nashorn(String parser, String code) throws ScriptException,NoSuchMethodEx
        ScriptEngineManager factory = new ScriptEngineManager();
        ScriptEngine engine = factory.getEngineByName("nashorn");
        engine.eval(parser);
        Invocable inv = (Invocable) engine;
        Object esprima = engine.get("esprima");
        Object tree = new Object();
        Object tokens = new Object();
        for (int i = 0; i < RUNS; ++i) {
            long start = System.nanoTime();
            tree = inv.invokeMethod(esprima, "parse", code);
            tokens = inv.invokeMethod(esprima, "tokenize", code);
            long stop = System.nanoTime();
            System.out.println("Run #" + (i + 1) + ": " + Math.round((stop - start) / 1e6
        }
        // System.out.println("Data is " + tokens.toString() + " and " + tree.toString())
    }
```

从代码可以看出，测试程序将执行Esprima的parse和tokenize来运行测试文件的内容，Rhino和Nashorn分别执行30次，在开始时候，Rhino需要1726 ms并且慢慢加速，最终稳定在950ms左右，Nashorn却有另一个特色，第一次运行耗时3682ms，但热身后很快加速，最终每次运行稳定在175ms，如下图所示

时间单位：ms											
Rhino	1726	1142	1195	1297	1100	953	965	932	931	959	961
Nashorn	3682	1623	930	669	474	335	368	267	231	194	175



nashorn首先编译javascript代码为java字节码，然后运行在jvm上，底层也是使用invokedynamic命令来执行，所以运行速度很给力。

为何要用java实现javascript

这也是大部分同学关注的点，我认同的观点是：

1. 成熟的GC
2. 成熟的JIT编译器
3. 多线程支持
4. 丰富的标准库和第三方库

总得来说，充分利用了java平台的已有资源。

总结

新犀牛可以说是犀牛式战车，比Rhino速度快了许多，作为高性能的javascript运行环境，Nashorn有很多可能。

举例， Avatar.js 是依赖于Nashorn用以支持在JVM上实现Node.js编程模型，另外还增加了其他新的功能，如使用一个内建的负载平衡器实现多事件循环，以及使用多线程实现轻量消息传递机制；Avatar还提供了一个Model-Store, 基于JPA的纯粹的JavaScript ORM框架。

在企业中另外一种借力 Nashorn方式是脚本，相比通常我们使用Linux等shell脚本，现在我们也可以使用Javascript脚本和Java交互了，甚至使用Nashorn通过REST接口来监视服务器运行状况。

本文代码地址是：http://git.oschina.net/benhail/javase8-sample

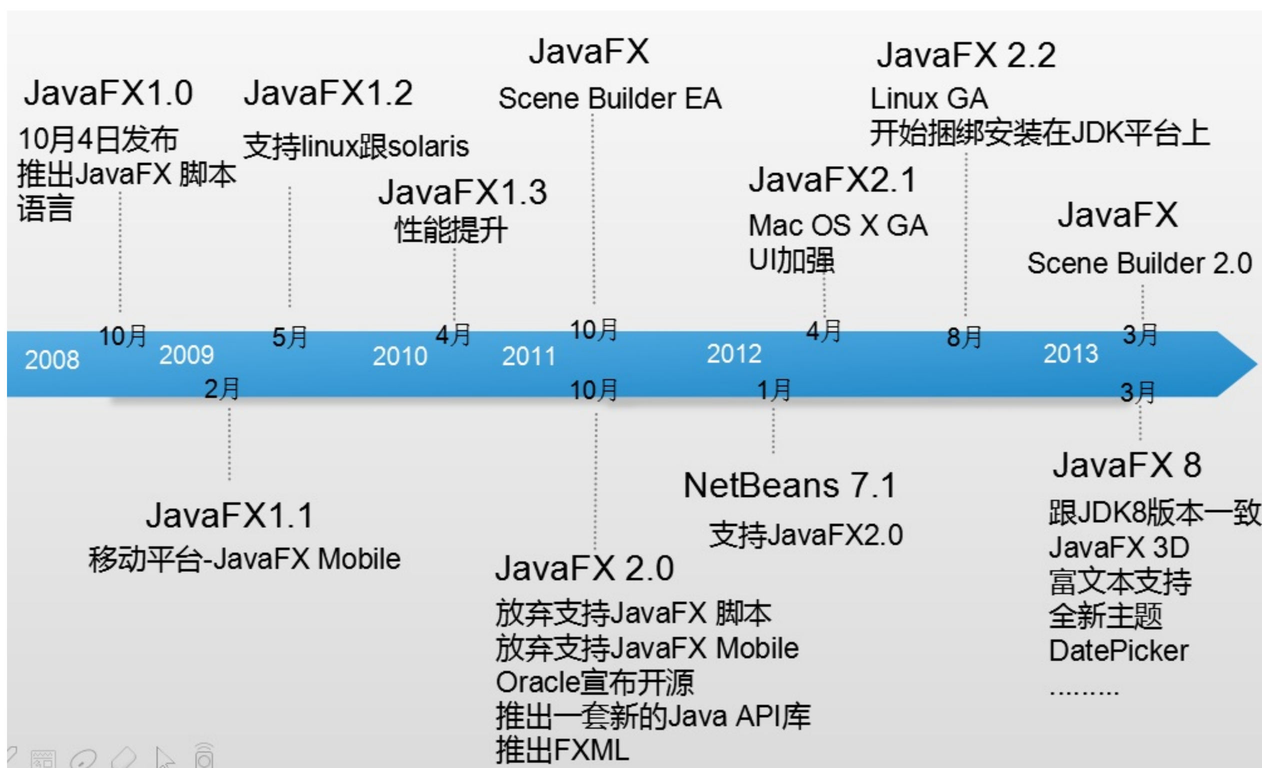
十三：JavaFX8新特性以及开发2048游戏

> 来源：Java 8新特性探究（十三）JavaFX 8新特性以及开发2048游戏

JavaFX主要致力于富客户端开发，以弥补swing的缺陷，主要提供图形库与media库，支持audio,video,graphics,animation,3D等，同时采用现代化的css方式支持界面设计。同时又采用XUI方式以XML方式设计UI界面，达到显示与逻辑的分离。与android这方面确实有点相似性。

JavaFX历史

跟java在服务器端和web端成绩相比，桌面一直是java的软肋，于是Sun公司在2008年推出JavaFX，弥补桌面软件的缺陷，请看下图JavaFX一路走过来的改进



从上图看出，一开始推出时候，开发者需使用一种名为JavaFX Script的静态的、声明式的编程语言来开发JavaFX应用程序。因为JavaFX Script将会被编译为Java bytecode，程序员可以使用Java代码代替。

JavaFX 2.0之后的版本摒弃了JavaFX Script语言，而作为一个Java API来使用。因此使用JavaFX平台实现的应用程序将直接通过标准Java代码来实现。

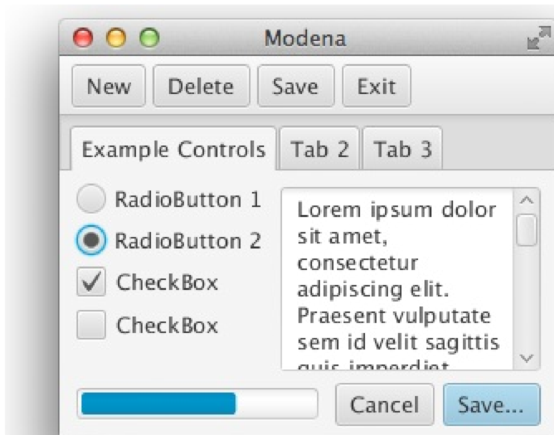
JavaFX 2.0 包含非常丰富的 UI 控件、图形和多媒体特性用于简化可视化应用的开发，WebView可直接在应用中嵌入网页；另外 2.0 版本允许使用 FXML 进行 UI 定义，这是一个脚本化基于 XML 的标识语言。

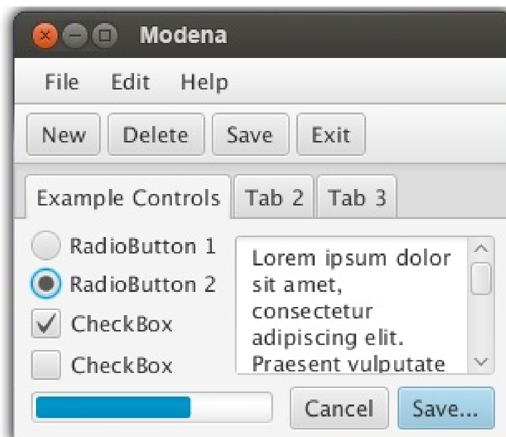
从JDK 7u6开始，JavaFx就与JDK捆绑在一起了，JavaFX团队称，下一个版本将是8.0，目前所有的工作都已经围绕8.0库进行。这是因为JavaFX将捆绑在Java 8中，因此该团队决定跳过几个版本号，迎头赶上Java 8。

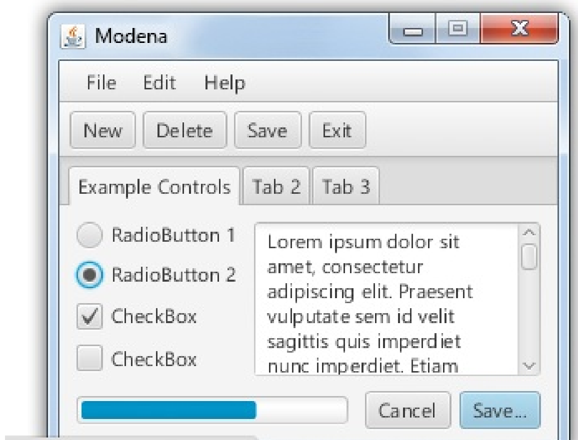
JavaFx8的新特性

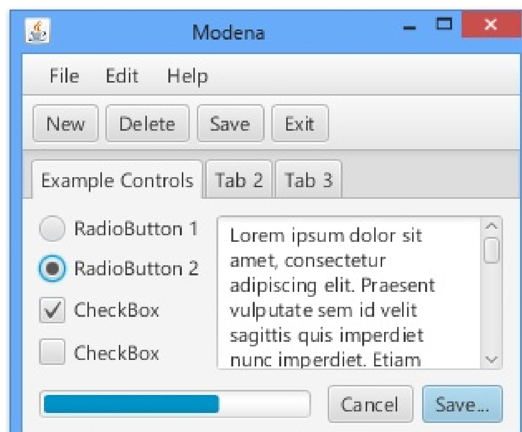
全新现代主题：**Modena**

新的Modena主题来替换原来的Caspian主题。不过在Application的start()方法中，可以通过setUserAgentStylesheet(STYLESHEET_CASPIAN)来继续使用Caspian主题。





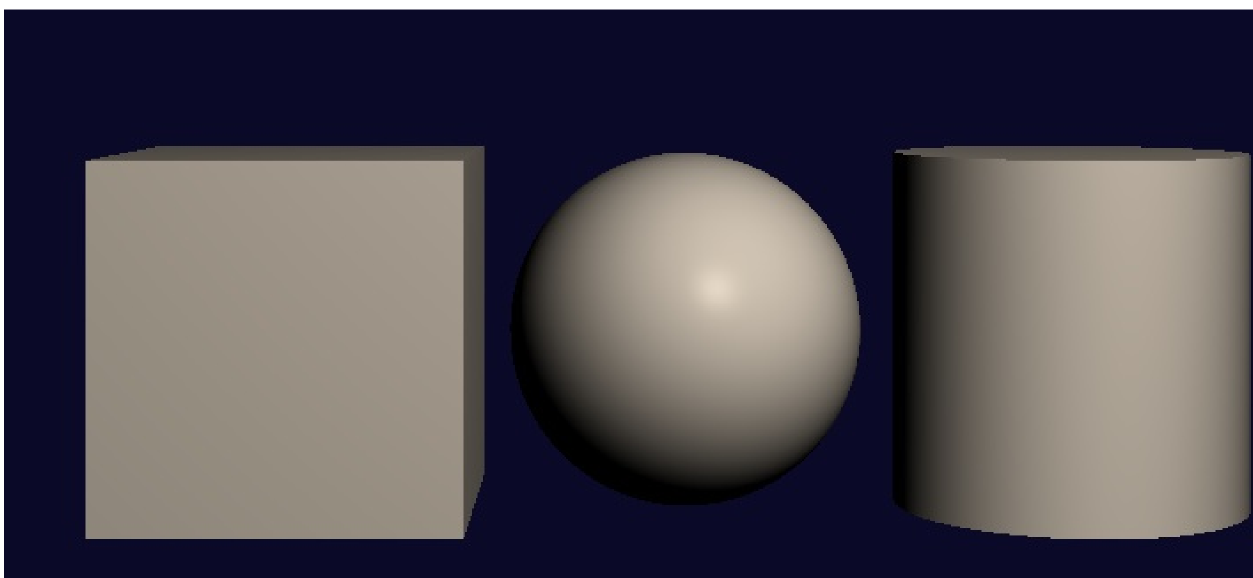




参考http://fxexperience.com/2013/03/modena-theme-update/

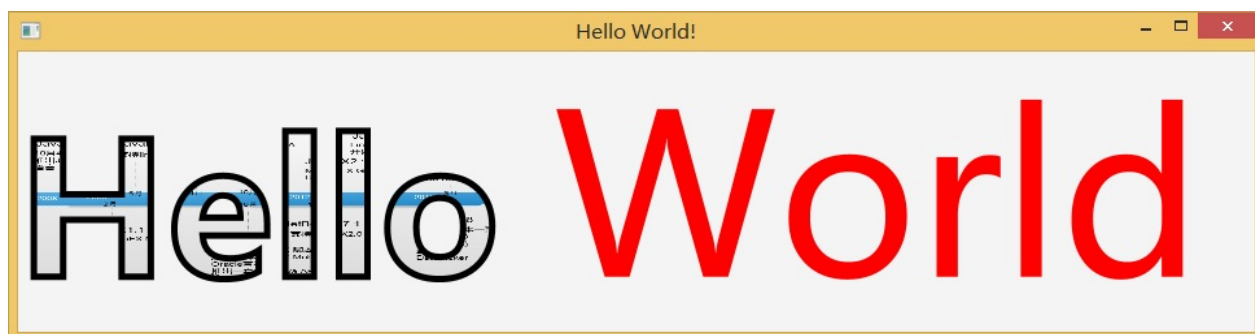
JavaFX 3D

在JavaFX8中提供了3D图像处理API，包括Shape3D (Box, Cylinder, MeshView, Sphere子类),SubScene, Material, PickResult, LightBase (AmbientLight 和PointLight子类),SceneAntialiasing等。Camera类也得到了更新。从JavaDoc中可以找到更多信息。

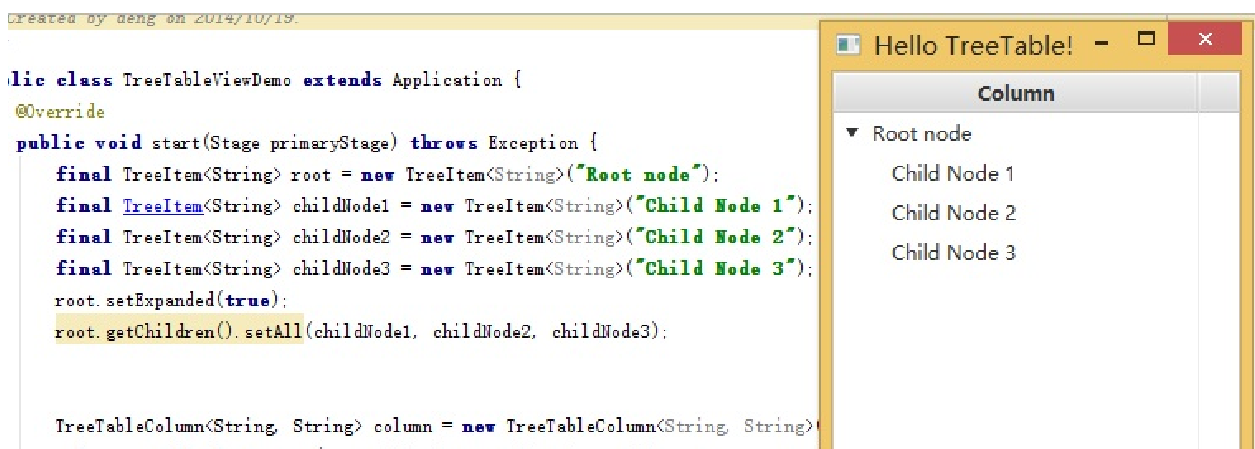


富文本

强化了富文本的支持

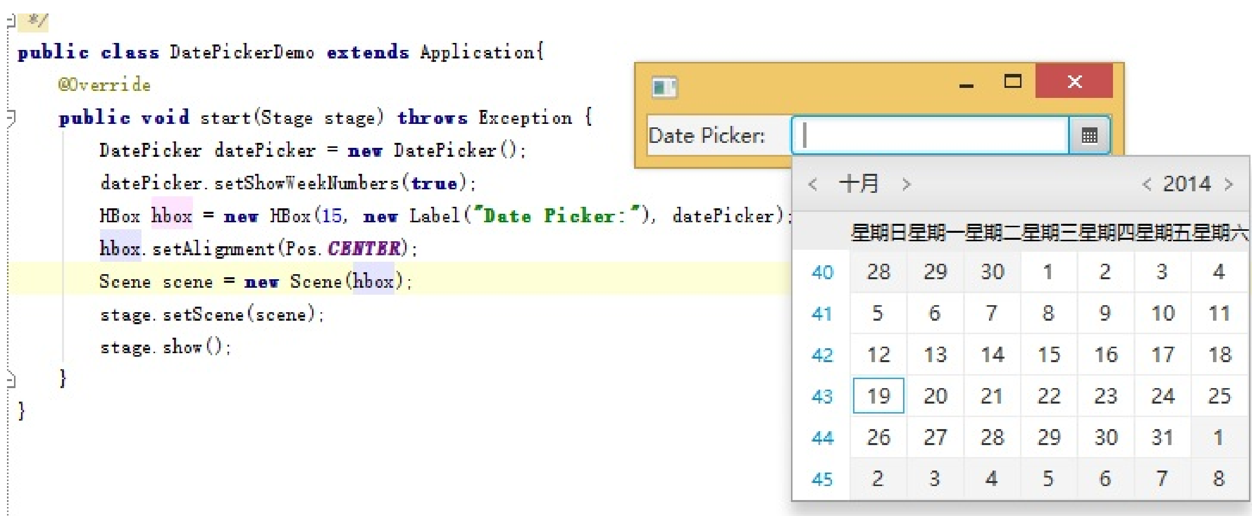


TreeTableView



日期控件**DatePicker**

增加日期控件



用于 **CSS** 结构的公共 **API**

1. CSS 样式设置是 JavaFX 的一项主要特性
2. CSS 已专门在私有 API 中实现（com.sun.javafx.css 软件包）
3. 多种工具（例如 Scene Builder）需要 CSS 公共 API
4. 开发人员将能够定义自定义 CSS 样式

WebView 增强功能

1. Nashorn JavaScript 引擎 https://blogs.oracle.com/nashorn/entry/open_for_business
2. WebSocket http://javafx-jira.kenai.com/browse/RT-14947
3. Web Workers http://javafx-jira.kenai.com/browse/RT-9782

JavaFX Scene Builder 2.0

可视化工具，加速JavaFX图形界面的开发，下载地址

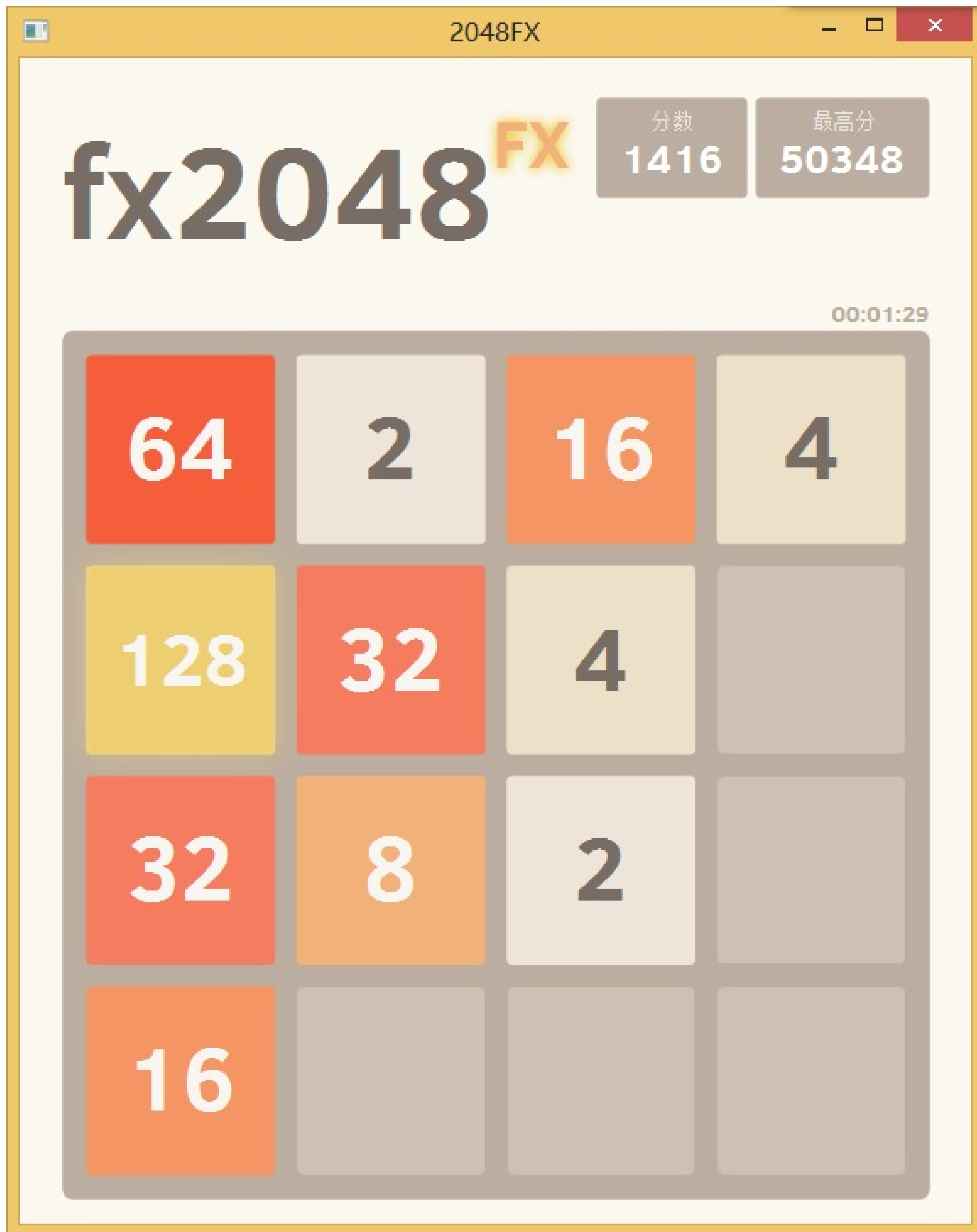
JavaFX Scene Builder如同NetBeans一般，通过拖拽的方式配置界面，待完成界面之後，保存为FXML格式文件，此文件以XML描述物件配置，再交由JavaFX程式处理，因此可減少直接以JavaFX编写界面的困難度。

JavaFX Scene Builder 2.0新增JavaFX Theme预览功能，菜单「Preview」→「JavaFX Theme」选择不同的主题，包括：

- Modena (FX8).
- Modena Touch (FX8).
- Modena High Contrast – Black on White (FX8).
- Modena High Contrast – White on Black (FX8).
- Modena High Contrast – Yellow on Black (FX8).
- Caspian (FX2).
- Caspian Embedded (FX2).
- Caspian Embedded QVGA (FX2).

JavaFX 8开发2048游戏

2048虽然不像前段时间那么火了，但个人还是非常喜欢玩2048，空闲时间都忍不住来一发，感谢 Gabriele Cirulli 发明了这了不起 (并且会上瘾)的2048游戏，因为是用MIT协议开源出来，各种语言版本的2048游戏横空出世，下图是用JavaFX 8来开发的一款2048。



所用到的技术

- Lambda expressions
- Stream API
- JavaFX 8
- JavaFX CSS basics

- JavaFX animations**fx2048**相关类的说明
- Game2048,游戏主类
- GameManager,包含游戏界面布局（Board）以及Grid的操作（GridOperator）
- Board,包含labels ，分数，grid ，Tile
- Tile,游戏中的数字块
- GridOperator,Grid操作类
- Location,Direction 位置帮助类
- RecordManager，SessionManager，纪录游戏分数，会话类

这里是源码地址，大家感兴趣的可以去学习下

总结

以上的相关源码都托管在 这里。

比起AWT和SWING，JavaFX的优势很明显，各大主流IDE已经支持JavaFX的开发了，最佳的工具莫过于NetBeans，且随着lambda带来的好处，JavaFX的事件处理简洁了不少，以前需要写匿名函数类。另外JavaFX开源以来，JavaFX的生态环境也越来越活跃了，包括各种教程，嵌入式尝试，还有一些开源项目，比如：ControlsFX，JRebirth，DataFX Flow，mvvmFX，TestFX 等等。还有JavaFX是可以运行在Android和ios上面，这个很赞！

好了，总结到这里也差不多了，在RIA平台上面，有HTML5、Flex和微软的Sliverlight，JavaFX能否表现优秀，在于大家的各位，只要我们多用JavaFX，那么JavaFX也会越来越优秀，任何语言都是这样, THE END .